

随书DVD含本书源码、学习环境及资料

Broadview
www.broadview.com.cn



Professional Embedded Software Development

专业嵌入式软件开发

全面走向高质高效编程

李云◎著



電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Professional Embedded Software Development

专业嵌入式软件开发

全面走向高质高效编程

这是一本全面讲解实时操作系统实现原理的书。为了使读者获得最佳的学习效果，作者为本书量身打造了可在Windows和Linux操作系统上直接运行的ClearRTOS“实时”操作系统，并在书中详解了所有实现细节。

这是一本介绍嵌入式软件开发所需掌握工具的书。作者从实用的角度介绍了gcc编译器、binutils工具集、ld链接器和gdb调试器，并花了较大的篇幅帮助读者成为Makefile的专家。

这是一本带领读者实践如何构建高质高效软件开发方法的书。书中通过展示如何将单元测试框架、静态分析、动态分析和性能分析整合到开发环境中这种方式，阐述了作者的“以单元测试为中心”和“无缝整合”思想。

这是一位饱尝自学嵌入式软件开发痛苦的工程师在软件行业积累了12年后，与读者分享心得的一本书。书中就软件设计、编程习惯和质量保证等内容与读者进行了交流。

上架建议：嵌入式开发



ISBN 978-7-121-14783-8





策划编辑：张春雨
责任编辑：葛　娜
封面设计：侯士卿

定价：108.00元（含DVD光盘1张）

Professional Embedded Software Development

专业嵌入式软件开发

全面走向高质高效编程

李云◎著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书分为 6 篇。硬件篇就嵌入式软件开发所需掌握的处理器概念进行了介绍。工具篇对 make、gcc 编译器、bintuils 工具集、ld 链接器和 gdb 调试器进行了讲解，其中对 make 这一嵌入式开发环境的全能管家进行了精辟的介绍，致力于帮助读者成为 Makefile 方面的专家。编程语言篇致力于让读者更深入地理解 C 编程语言。操作系统篇通过循序渐进的方式介绍 ClearRTOS 的设计与实现，使得读者能透彻地理解操作系统的关键概念和实现原理。设计篇和质量保证篇通过实践的方式逐步展开讲解，以帮助读者获得一些实用的设计原则、最佳实践和一套有效的质量保证方法论。

本书适合嵌入式软件开发领域的新手和在工作中碰到瓶颈的老手阅读。阅读本书要求读者已掌握 C 编程语言和基本的 UML 知识。

图书在版编目（CIP）数据

专业嵌入式软件开发：全面走向高质高效编程 / 李云著. —北京 ：电子工业出版社，2012.1
ISBN 978-7-121-14783-8

Ⅰ. ①专…　Ⅱ. ①李…　Ⅲ. ①程序设计　Ⅳ.①TP311.1

中国版本图书馆 CIP 数据核字（2011）第 206049 号

策划编辑：张春雨
责任编辑：葛　娜
印　　刷：北京东光印刷厂
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1092　1/16　印张：40　字数：1020 千字
印　　次：2012 年 1 月第 1 次印刷
印　　数：4000 册　定价：108.00 元（含 DVD 光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

我于2000年第一次接触嵌入式软件开发工作，那时和很多入门者一样，因为找不到全面、易懂、深入的读物，也没有人指导，因而遭遇了极大的自学痛苦。即使在今天，学习嵌入式软件开发似乎仍困难重重，这从我的博客空间不时有网友发私信询问如何学习可以看出。

我也曾被网友要求推荐学习嵌入式软件开发的好书。但当我以"嵌入式"关键字在网上书店进行搜索时，所获得的书大部分与Linux、Windows CE、Android和ARM有关。在我看来，网友并不是让我帮助他选择Linux还是Windows CE，ARM还是x86，而认为他希望获得一本学习通用原理和方法的书，因此不敢贸然推荐。基于这种现状，我萌发了写一本既能指导新手入门，又能帮助老手获得突破的书。读者手上拿的正是这本书！本书的创作始于2009年6月，历时2年后于2011年下半年面市。

在本书的创作之初，我问自己：这本书应当包含哪些内容呢？或许可以根据自己过去十多年所经历并克服的成长痛苦进行编排！

嵌入式软件开发是一种软硬件结合非常紧密的职业，对工程师的能力要求自然也就高了。刚开始学习嵌入式软件开发时，最困难的莫过于学习操作系统原理和处理器方面的知识，所以本书必须包含这两方面的内容。讲解操作系统原理如果以Linux、Windows CE等成熟的操作系统为素材并不好，因为它们太大，很容易让人"只见森林不见树木"，也容易让人望而生畏而失去学习的兴趣和信心。从软件开发的角度来看，操作系统的概念和实现原理一旦掌握，不论基于哪一个操作系统做开发都只是调用不同的函数而已。为了让读者获得最好的学习体验，我为本书设计了一个实现简洁、完整的"实时"[①]操作系统——ClearRTOS，通过渐进式的方式细致地讲解操作系统的概念和实现原理。至于处理器方面的知识，本书没有针对某一具体处理器，而是就编程方面所需的通用知识进行了介绍。对这些通用知识的掌握，将使得处理器对于读者不再那么神秘。

学习嵌入式软件开发的另一大困难是实践问题，本书必须帮助读者解决这一问题。对于很多初学者来说，为了实践而购买一块开发板的学习成本偏高。值得欣喜的是，读者学习本书并不需要购买开发板，而只需要有一台安装于x86或x86-64（包括Intel 64[②]和AMD 64）处理器

---

① "实时"打上引号是因为ClearRTOS在Cygwin环境和Linux操作系统中无法直接接管处理器的中断，所以无法实现在中断返回的过程中完成任务切换的功能。这是与真正的嵌入式操作系统唯一的区别。

② IA-64不同于Intel 64，后者是指x86-64，详情参见http://en.wikipedia.org/wiki/x86-64。

上的 Windows 或 Linux 操作系统的计算机，对于大多数读者来说这样的学习环境就在身边。另外，软件开发工程师有一个特点，对于自己能修改和调试的代码更具学习兴趣，通过这种方式学习的效果也更佳。本书的所有代码（包括 ClearRTOS）都被设计成能在 Cygwin 环境③和 Linux 操作系统上编译、调试和运行，所以本书完全迎合工程师的这一学习偏好。总的说来，实践性强是本书很突出的一个特色。

掌握开发所需的工具是学习嵌入式软件开发的又一大挑战，本书在这方面也花费了大量笔墨。与非嵌入式软件开发采用集成开发环境不同，嵌入式软件开发大多是基于命令行的。软件开发工程师除了进行编码工作，还需要能驾驭自己的编译环境并运用其他的开发工具辅助开发工作。本书的工具篇以来自 GNU 的工具为例帮助读者战胜这一挑战。值得强调的是，其中花了很大的篇幅帮助读者成为 Makefile 方面的专家。

如果读者只想入门，那么掌握操作系统、处理器和必要的工具就足够了。但如果想获得突破，以实现高质高效地从事软件开发工作显然不够，还必须理解软件设计的重要性，并借助一定的质量保证方法论来提高工作质量和效率。软件设计和质量保证方法论是业内比较抽象和高级的话题，为此本书在设计篇和质量保证篇通过实践的方式逐步展开讲解，以帮助读者获得一些实用的设计原则、最佳实践和一套有效的质量保证方法论。

总而言之，本书从知识、工具、方法和思想这四大方面全面讲解如何专业地从事嵌入式软件开发，致力于帮助读者全面走向高质高效编程。

读者阅读本书之前，需要掌握 C 编程语言和基本的 UML 知识④。如果有使用 Linux 操作系统的基础经验，对学习本书也会有小小的帮助⑤。尽管本书是针对嵌入式领域的，但书中的很多思想和方法适用于整个软件行业。

本书结构

全书分为 6 大篇共 33 章，读者可以通过浏览书的目录以进一步了解各篇所涵盖的内容。

硬件篇就嵌入式软件开发所需掌握的处理器概念进行了介绍，并通过介绍电路信号的完整性问题告诉读者，嵌入式产品的质量不是软件质量单方面能保证的。

工具篇介绍了提高嵌入式软件开发效率所需掌握的工具。make 作为嵌入式开发环境的全能管家，在本篇中花了较大的篇幅对其进行精辟的介绍。此外，gcc 编译器、binutils 工具集、

---

③ Cygwin 是一个开源项目，实现了在 Windows 操作系统上虚拟 Linux 操作系统的环境。

④ UML 是“Unified Modeling Language”的缩写，即统一建模语言。如果需要，读者可以以“跟我学 UML”为关键字在我的博客中查找所需的学习资料。

⑤ 本书并不需要读者有丰富的 Linux 操作系统使用经验，只需掌握十几个命令就行了。即使读者第一次接触 Linux，对于学习本书也不会有困难。

ld 链接器和 gdb 调试器都在本篇中涵盖了。对于工具的介绍是基于实用的角度展开的，而不是“大全”。

编程语言篇致力于让读者更深入地理解C编程语言。其中对程序的结构、ABI/EABI、volatile 关键字进行了讲解，这几方面的知识在非嵌入式软件开发中并不需要深入了解，但在嵌入式软件开发中却是必须掌握的。本篇还通过分析一个因混淆指针和数组所导致的问题，指出开发活动中容易忽视的一个认识盲点，并提出了预防这类问题的终极方法。

设计篇解释了为什么设计是软件产品的质量之本，还介绍了作者常用的设计原则及所倡导的软件设计思想和一些最佳实践。设计思想包括：平台与框架开发、可查错性设计、可开发性设计；最佳实践则覆盖模块管理和错误管理。

操作系统篇通过循序渐进的方式介绍 ClearRTOS 的设计与实现，使得读者能透彻地理解操作系统的关键概念和实现原理。读者掌握这篇的内容，有助于轻松地在实时 Linux、VxWorks、Windows CE 等各种实时操作系统上从事软件开发工作。

质量保证篇关注于如何通过质量保证方法论来获得高质量的软件产品，也探讨了工程师的编程习惯对软件质量的影响。本篇中强调了单元测试这一被忽视的质量保证方法的价值，并通过设计实用的单元测试框架展示如何在项目中实施它。本篇中还展示了如何将代码覆盖、静态分析、动态分析和性能分析无缝地整合到开发环境中，以及阐述了“以单元测试为中心”和“要素有形化”质量保证方法论设计思想的具体含义。

致谢

本书是我的处女作，能与读者见面离不开很多人的支持和帮助。首先，感谢我的妻子和女儿。正是在妻子的提议下，我从写博客开始扬起了本书的写作之帆。女儿则是我的开心果，给我的写作之路带来了很多的乐趣，让我得到更多的放松机会。

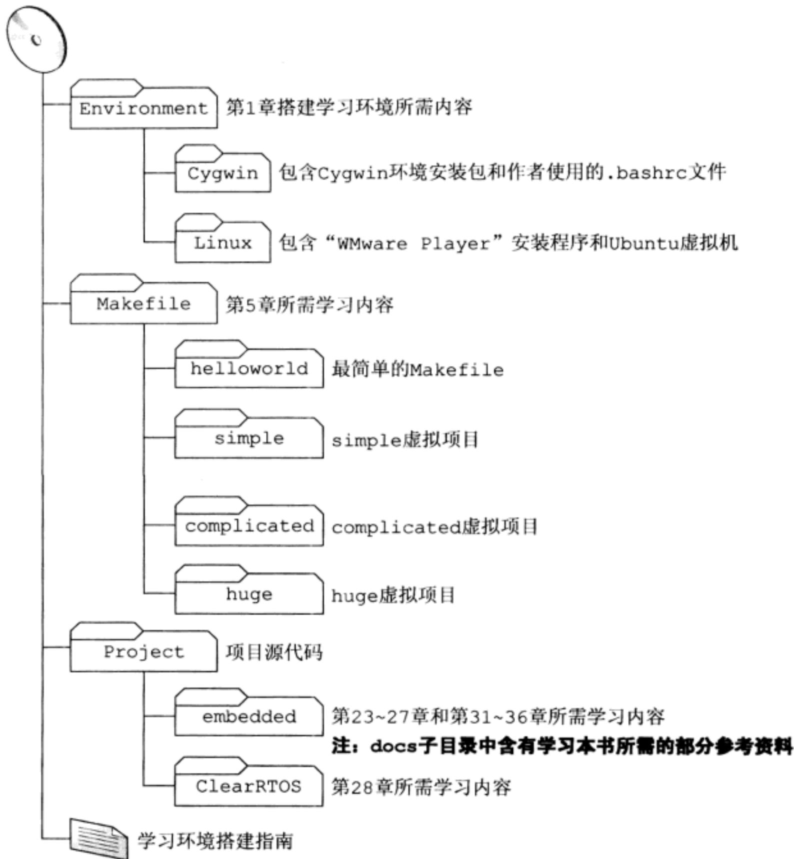
感谢我的朋友及职业生涯中的上司和同事，正是他们给我机会，或鼓励，或帮助，让我一路积累，才有可能完成本书的创作。他们包括但不限于：庞惠民、章佳欢、刘伟民、夏青、于善成、范鹏、罗延庭。

感谢 51CTO 博客的同仁，他们的幕后支持让我坚持了下来。广大 51CTO 博友的期待也激励着我努力地写好本书。

感谢电子工业出版社的策划编辑张春雨，他的出现加速了本书的面市，也给我吃了一颗将书写到底的“定心丸”。与他交流写作方面的话题让我感受到了什么是隔行如隔山，他对出版行业的专家意见和追求满分的精神让本书增色不少。

最后，再一次感谢我的妻子和好友于善成，两位预读了本书并提出了自己的真知灼见，使得本书更简练、严谨和更具可读性。

附书光盘内容介绍



****

书中如果出现错误，请先接受作者的致歉，如能来信告知那将不胜感激。错误一旦发现会通过我的博客第一时间通知其他读者。读者在学习中如需帮助，可以通过技术圈发帖讨论。

联系方式

邮箱：　yunli.book@gmail.com

微博：　weibo.com/UltraEmbedded

博客：　yunli.blog.51CTO.com

技术圈：　g.51CTO.com/UltraEmbedded

李　云

2011-07-04

目 录

硬件篇

工具篇

编程语言篇

设 计 篇

操作系统篇

质量保证篇

硬件篇

处理器是嵌入式系统的核心部件，对于处理器的了解不仅有助于理解编程语言和掌握计算机的体系结构，还有助于我们编写出更专业的程序。在第 1 章我们将一同探讨嵌入式软件开发中所需掌握的处理器相关的知识。

非嵌入式软件开发几乎不用担心运行软件的主机存在硬件问题，但做嵌入式软件开发就不一样了。在嵌入式系统中，如果对一些缺陷一味地从软件方面下工夫查错，将永远无法获得答案。在第 2 章就硬件中的信号完整性问题进行了介绍，并从软件开发的角度提出了一些应对措施。了解信号完整性问题，有助于我们更全面地分析和解决所面临的复杂缺陷。



第 1 章 处理器的基本概念

在嵌入式软件开发中，我们与处理器的距离更近。正因如此，从事嵌入式软件开发需要我们掌握更多与处理器相关的知识。无论是什么类型的处理器，它们的基本概念都相似。在此对市面上所有的处理器进行逐个介绍显然不切实际，但可以通过介绍基本概念的方法使我们对之不那么陌生。

对于处理器基本概念的掌握不仅有助于开展嵌入式软件开发工作，还有助于我们更加深入地理解编程语言和掌握计算机体系结构。读完本章后读者会发现，编程语言中的一些概念（比如字节序、边界对齐等）正是源于处理器的。

1.1 区分微处理器与微控制器

嵌入式系统的处理器大多是微控制器（microcontroller），微控制器不同于微处理器（microprocessor），它是指在同一块芯片内除了中央处理单元（CPU）之外还集成了部分内存和外设（参见 1.4 节）。图 1.1 大致地示例说明了微处理器与微控制器的区别。集成于微控制器内的内存和外设我们分别称之为“片内内存”和“片内外设”，否则称之为“片外内存”和“片外外设”。

我们常用的台式机和笔记本电脑中的处理芯片属于微处理器。很显然，微处理器提供高速的总线以实现与外部的内存和外设进行交互。协调处理器的高速总线与速度较之更慢的外设，需要通过芯片组来完成。

嵌入式系统大多是微控制器的原因，是为了节约成本和节省功耗。在实现相同功能的前提下，将大量的芯片集成在一块芯片内的制造和使用成本，以及功耗都更低。另外，由于微控制器内集成了大量的外设，使得嵌入式系统的硬件设计得到了极大的简化。

本书将微处理器和微控制器统称为处理器。这是因为从编程的角度来看，微处理器与微控制器其实没有区别。

1.2 寄存器

处理器（这里特指中央处理单元，或 CPU）是通过寄存器来运行程序和加工数据的。不同

的处理器所包含的寄存器数量和名称有所不同，但处理器寄存器的功用却大同小异。可以说寄存器内的值（的变化）决定了处理器的行为。

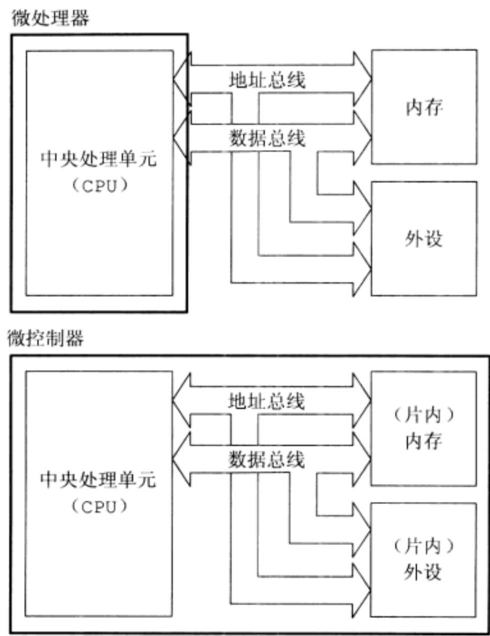


图 1.1

寄存器可以分为两大类——通用寄存器（General Purpose Register，GPR）和浮点寄存器（Floating Point Register，FPR）。通用寄存器的作用包括执行指令、进行整型数据和逻辑运算，而浮点寄存器则用于运算具有小数点的数据。

在通用寄存器中一定存在程序计数器，简写为 PC（Program Counter）。这一寄存器用于告诉处理器下一条执行指令在地址空间中的位置。注意，这里的地址空间不只指内存空间，还可以包含像闪存这样的空间。程序计数器也被称为指令指针（Instruction Pointer，IP），在 x86 处理器中就是使用这一称呼的。在本书中，程序计数器和指令指针是两个可互换的术语。

从处理器的角度来看，程序的运行是借助程序计数器来做导航的。每执行一条指令程序计数器中的值就会发生变化，变化的程度与所执行的指令有关。当碰到跳转和调用指令时，程序计数器内的值将发生跳变；否则程序计数器的值只是增加几个字节。

在通用寄存器中还存在栈指针寄存器 SP（Stack Pointer），栈的作用在 1.6 节和 10.4.1 节有更为详细的描述。在 x86 处理器上，栈指针被称为 ESP（Extended Stack Pointer）。

除了以上两个寄存器，通用寄存器中还有一些其他寄存器，其功能无外乎与变量处理、参数传递有关。

具有浮点运算单元的处理器还设计有专门的浮点寄存器，通过这些寄存器实现高效的浮点运算。

实际上，不光是中央处理单元存在寄存器，集成在微控制器内的外设也有寄存器。通过配置这些寄存器，可以控制外设的行为和工作方式。此外，这些寄存器在处理器的地址空间中占有相应位置，配置这些寄存器就是对这些空间根据芯片手册进行读写操作。

1.3 处理器是如何启动的

工程师大多喜欢寻根究底，因此处理器是如何启动的这一话题也总是让人着迷。不同的处理器尽管启动原理大致相同，但启动过程还是存在多样性。了解处理器是如何启动的就类似于我们了解 C 语言的入口函数是 main()一样，具有“导读性”意义。

当熟悉一个已经开发好的软件项目时，因为我们了解 C 程序的入口是 main()函数，所以可以从其着手“顺藤摸瓜”地了解整个软件的实现和大致架构。这一方法对任何功能的软件都适用。同样地，了解处理器的启动流程对于真正理解所使用的处理器也很有意义，这有助于我们更好地把握计算机的体系结构。

每块处理器在出厂时已固化好其寄存器的默认值，这些值决定了处理器上电（即给处理器供电）时刻的行为。程序计数器的默认值决定了处理器从哪一个具体地址去获得第一条需要执行的指令。为了解释方便，假设某一处理器程序计数器上电时的默认值是 0xFFFF0000。

那 0xFFFF0000 这一地址对应于哪一个具体的存储设备呢？假设第一条执行指令是放在闪存中的，处理器如何知道 0xFFFF0000 地址所对应的指令应从闪存中获取？

对于处理器来说，不论它的总线上挂接的是闪存、内存还是硬盘，它在启动时一无所知，我们需要通过硬件设计来告诉它存储第一条指令的外设，也就是说，处理器的第一条执行指令的地址是通过硬件设计来实现的（这是硬件工程师的工作内容）。

处理器一启动，就会从 0xFFFF0000 这个地址读取指令。读取第一条指令的同时，处理器会产生对应地址空间的片选信号，以使能位于 0xFFFF0000 地址处的存储器件。如果希望 0xFFFF0000 地址所对应的就是闪存的第一个字节，那么就需要通过硬件设计，将闪存的片选信号与处理器的 0xFFFF0000 地址所对应的片选信号相连，且通过恰当的地址线连接使得闪存的第一个字节就在 0xFFFF0000 处。也就是说，硬件设计需要完成地址与外设间的映射。

在大多数嵌入式系统中，处理器所取得的第一条指令应当属于引导加载器程序的一部分，而第一条指令的获取标志着引导加载器程序开始运行。关于引导加载器的更多细节请参见第 19 章。

1.4 输入与输出

除了中央处理单元和内存这两大部件外，另外一大部件是外设（peripheral）。外设是一个

非常宽泛的概念，既可以集成在微控制器芯片内（我们称为片内外设），也可以是挂在处理器总线上的外部芯片（我们称为片外外设）。外设的种类有很多，常见的有：用于实现以太网通信的外设；用于实现 RS232 串行通信的外设；用于实现 USB 通信的外设；用于实现存储的闪存外设；用于实现图像采集的外设；等等。中央处理器与外设的通信被称为输入与输出（Input/Output），简称为 I/O。

外设也像内存的存储单元那样通过地址进行区分，但它们的地址被称为 I/O 端口（I/O port）。每一个外设在处理器的地址空间中占用不同的 I/O 端口，处理器可以通过不同的 I/O 端口实现与对应外设的通信。处理器除了通过 I/O 端口进行通信外，另一个重要的手段是中断。

I/O 端口所在的空间被称为 I/O 空间，各种架构的处理器存在不同的 I/O 空间设计形式。其中一种 I/O 空间设计是将之设计成独立于内存所在的空间。在这种设计下，读写 I/O 端口需要使用与存取内存不一样的指令。从编程的角度来看，对 I/O 端口操作不能像操作内存那样直接使用 C 语言中的指针完成，而是需要调用相应的函数，这些函数内封闭了 I/O 端口的操作指令。另一种 I/O 空间设计是将之设计成与内存在同一个地址空间中，它也被称为内存映射 I/O 空间。从编程的角度来看，内存映射 I/O 空间的端口操作与访问内存是完全一样的。我们熟悉的 x86 处理器既有独立的 I/O 地址空间（大小为 64KB），也有内存映射 I/O 空间。但像 ARM、PowerPC 这样的处理器完全采用内存映射 I/O 空间。

不同类型的外设所占用 I/O 空间的大小也不同。比如，总线型的 NOR 闪存将占用一整片的地址空间，空间大小与闪存的容量是一样的；而像串口这样的外设，只占用少数几个 I/O 端口。

外设也像处理器那样存在配置和数据寄存器，不同的配置寄存器值使得外设的行为有所不同。I/O 端口实际上就是外设的寄存器在处理器 I/O 空间的地址。I/O 端口除了存在地址外，还存在大小之分。有的端口一次操作是以单字节为单位的，而有的端口则是以 4 字节为单位的。

当我们对 I/O 端口进行读写操作时，实际上是对外设的寄存器进行读写。硬件工程师在进行硬件设计时，需要从硬件的角度实现这些映射关系。他们需要考虑片选、译码及时序，使得外设能被无缝地映射到处理器的 I/O 空间中。从软件工程师的角度来看，我们只管对不同的 I/O 端口进行读写就行了。

对外设的 I/O 端口写什么、如何读是在外设的芯片手册中指定的。因此，要正确配置外设并与之交互，离不开熟读外设的芯片手册，这是嵌入式软件驱动开发的一项基本功。

1.5　指令与数据

严格说来，指令也是数据，是告诉处理器“干什么”的特殊数据。为了需要，我们将指令这一特殊的数据独立出来，而将数据的概念缩小为不包括指令的数据。

处理器并不理解任务和进程是什么，它只知道指令和数据；它也不知道什么是函数和变量，

因为这些只是C语言中的概念。

C程序被编译好了以后，编译器会将程序的指令和数据以段的形式分别组织（参见9.1节）。指令中所嵌入的地址信息用于告诉处理器需要加工哪一地址处的数据，地址信息的嵌入工作是在编译程序时由编译器完成的。

1.6 中断

处理器一方面运行位于内存中的指令并加工相应的数据，另一方面也得关心其周围所发生的事件。处理器关心“外部世界”的方式有两种。一种是“漠不关心”，完全将这种任务交给应用程序（即程序员）。应用程序则通过轮询的方式不断地查询外设是否有事务需要处理。轮询是通过读取外设的配置寄存器来实现的。

另一种处理方式则更为“贴心”，当处理器得知外设有事件需要处理时，暂停正在运行的指令流，并立即切换到另一种工作模式，即中断模式。引入中断的好处是，可以通过中断驱动的方式避免使用查询这种耗时的方法，从而提高处理器的处理能力。

当中断发生时，中断服务程序会被调用。中断服务程序可以理解为C语言中的一个函数，在处理器的初始化阶段通过某种形式与特定的中断绑定在一起。

为了使用处理器的中断功能，外设需要使用一根中断信号线，并将这一信号线与处理器的中断输入管脚连接在一起，当然这是硬件工程师要做的事。当外设有数据需要处理器处理时，需要通过主动变换中断线上的电平，产生中断信号进行通知。之前，外设还得在其状态寄存器中设置对应的值，以便处理器上的中断服务程序在处理中断时，可以通过状态寄存器了解中断的具体细节。比如，获知是接收数据的中断，还是发送数据的中断，或出错中断。

为了支持中断，处理器方面也得做必要的准备。需要让外设工作之前初始化好处理器的中断控制器，并安装好对应的中断服务程序或称为ISR（Interrupt Service Routine）。通过中断控制器可以设置中断的触发类型、开关中断等。

中断服务程序的实现通常需要包含如下几步操作。

（1）从外设读取中断状态寄存器的值以了解这次中断的原因是什么，以便做相应的处理。

（2）为了告诉外设处理器已经处理完了所需做的工作，处理器需要通过一定的方式通知外设芯片。这种方式就是向外设芯片的寄存器中的某一位写入一个值。比如，可能写入1表示清中断。当外设收到了处理器的清中断请求后，就会驱动中断线使其处于中断无效的电平状态[①]。

（3）清除处理器的中断信号标识。处理器中往往也会保存外部中断信号是否发生过这样的

① 这是针对电平触发类型的中断而言的，对于沿触发类型的中断，外设并不需要这么做。

标识，当我们处理完了外设芯片的中断时，需要清除处理器上的标识，为下一次中断的到来做准备。清除外设的中断信号必须发生在清除处理器的中断标识之前，否则会造成重复中断。

中断还存在触发方式问题。有两种触发方式，一种是电平触发；另一种是沿触发。电平触发是指通过电平的高低表示外设是否发出了中断信号，而沿触发则是通过中断线上的电平的升或降来表示的。对于沿触发的中断又存在两种方式。一种是中断线从低电平变为高电平，我们称为上升沿触发；另一种是中断线从高电平转换为低电平，我们称为下降沿触发。概括起来，中断的触发方式有电平触发、上升沿触发和下降沿触发，图 1.2 示例说明了这三种方式的中断有效期（或点）。

当处理器进入中断模式时，需要保存中断时刻处理器的所有寄存器的值，以便中断服务程序执行完后还能恢复到中断之前的状态并继续运行②。之所以存在这一保存操作，是因为处理器的寄存器资源在中断模式和非中断模式之间是共享的。这种处理方式的效果就是，处理器处理完中断后非中断模式的程序并不知道有中断发生过，好处是我们编写非中断程序时根本不需要考虑中断何时会发生。这正是我们所希望的，因为它使编程工作简化了。

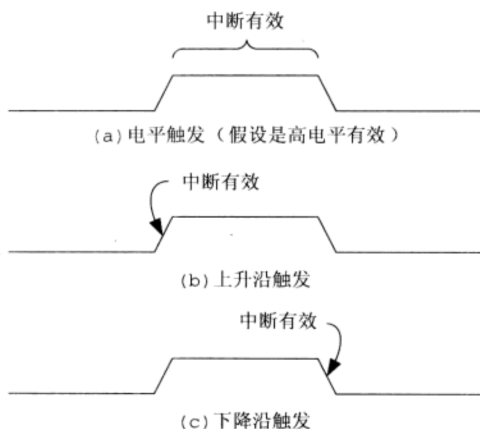


图 1.2

要保存中断之前处理器所有寄存器的值就得使用到栈。处理器处于中断状态的栈可以来源于两种：一种是当中断发生时，直接将需要保存的内容放入到当前正在运行程序的栈上，即中断状态没有独立的栈；另一种则是为中断状态提供特定的栈。当中断发生时，有一段代码被执行以便将栈切换到这一特定的栈。对于栈的作用和进一步解释，请参见 9.2 节和 10.4.1 节。

有的中断与外设是相关的，这类中断被称为硬中断。硬中断的特点是外设一定要用到处理器的中断信号线。有的中断与硬件毫无关系，是通过执行处理器的指令触发的。这类中断被称为软中断或陷阱（trap）。

中断存在优先级之别。在多个中断同时出现的情形下，高优先级的中断将先获得被处理的机会。如果高优先级的中断出现时，处理器正在处理一个低优先级的中断，低优先级的中断可被再一次打断而使得高优先级的中断获得被处理的机会。或者说中断存在“嵌套”功能。

在 1.5 节谈论指令与数据时提到，处理器的眼中并没有任务的概念，因此不存在“是中断的优先级高？还是任务的优先级高？”这么一说，因为它们根本就不在同一个概念域中。中断

② 可能根据需要只需保存部分寄存器，这与操作系统和处理器的实现有关。

永远具有比非中断模式运行的程序（包括任务）更高的优先级。

1.7　字节序

大多数处理器中内存是可以以字节为单位进行寻址的，当数据类型（比如 int、long）大于 1 个字节时，其所占用的字节在内存中的顺序存在两种模式，分别是小端模式（little endian）和大端模式（big endian）。小端模式是低位字节放在低地址，而大端模式则是高位字节放在低地址。

在 32 位处理器上，一个类型为 int 的 module_id 变量占用 4 个字节的内存。假设 module_id 位于 0x10000 内存地址处，则在小端模式的处理器上其各字节序如图 1.3 所示，图 1.4 所示是在大端模式的处理器上各字节序。

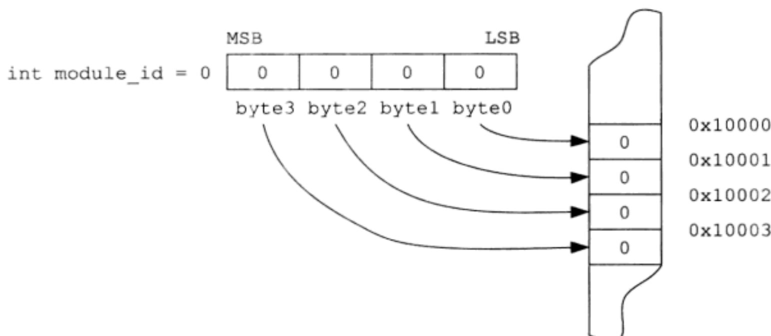


图 1.3

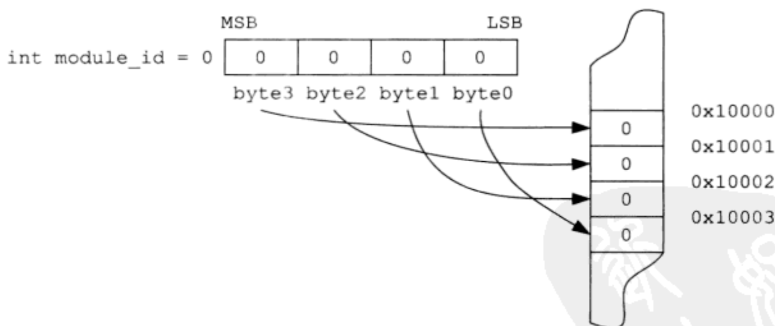


图 1.4

图中的 LSB 是指最低有效位(Least Significant Bit)，MSB 是指最高有效位(Most Significant Bit)。

为了理解字节序的重要性，让我们看一看图 1.5 的示例程序。如果在小端模式的处理器上运行该示例程序，其结果会是正确的，即在 foo()函数中打印出的模块标识是 3，而在 main()函数中打印出来是 4。但是，在大端模式的处理器上 main()函数输出的结果却是错误的。

```
#include <stdio.h>

void foo (short *_p_module_id)
{
   printf ("foo (): module ID is %d.\n", *_p_module_id);
   *p_module_id = 4;
}

int main ()
{
   int module_id = 3;
   foo ((short *)module_id);
   printf ("main (): module ID is %d.\n", module_id);
   return 0;
}
```

图 1.5

下面，我们分析一下为什么同一程序在不同字节序的处理器上运行结果会截然不同。图 1.6 示例说明了在小端模式下 main()调用 foo()时 module_id 变量中值的变化过程。注意，foo()函数的参数是以指针的形式传递的，也就是说，在 main()中 module_id 变量的起始地址如果是 0x10000，那么传入到 foo()函数中后_p_module_id 变量指向的地址也是 0x10000。

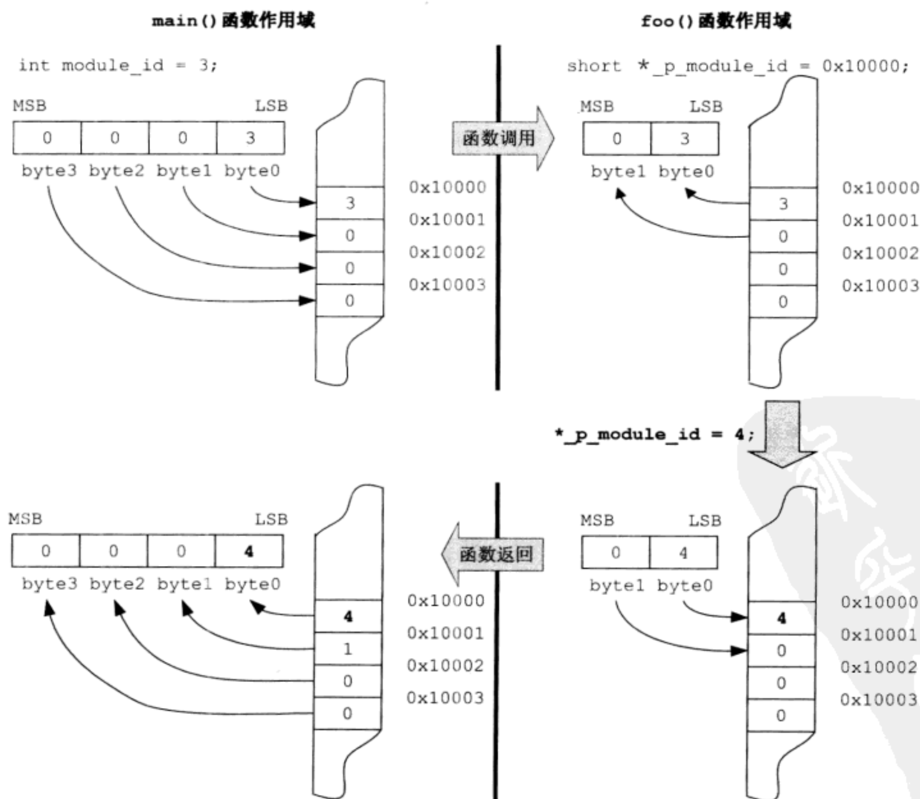


图 1.6

从图中可以看出，在小端模式下并不存在问题，这与示例程序的运行结果是相吻合的。图 1.7 则示例说明了在大端模式下的情形，从图中读者可以了解为什么会出现问题。

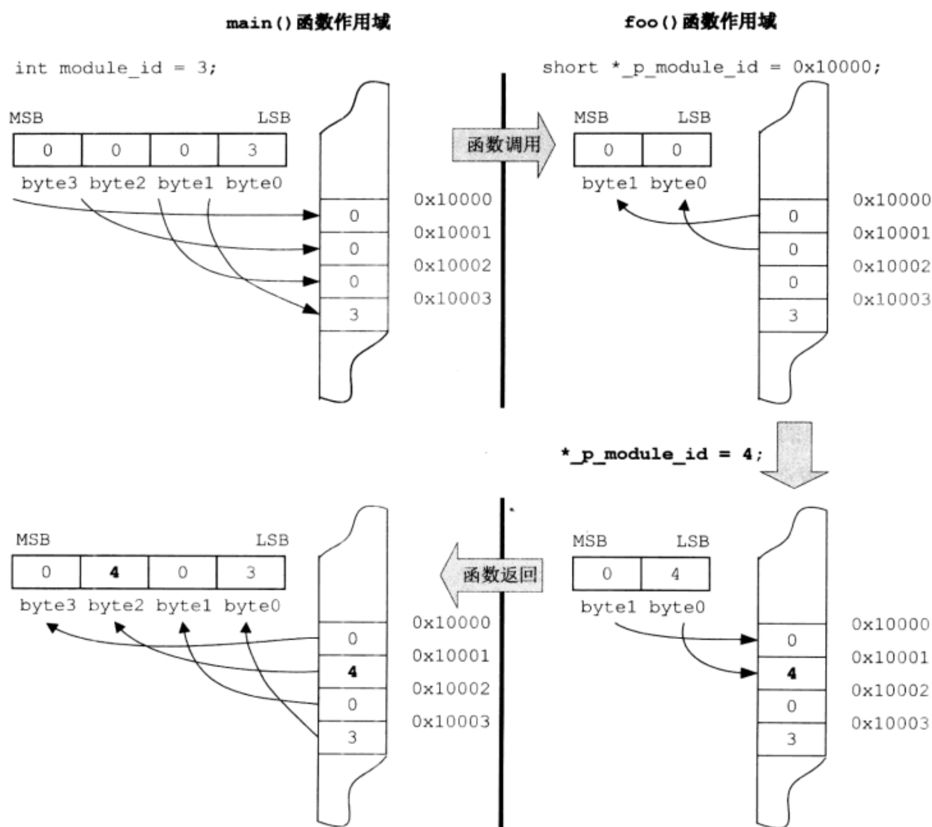


图 1.7

这一示例程序告诉我们，由于处理器两种字节序模式的存在，为了避免所编写的程序存在移植性问题，在软件开发过程中指针应当严格按照所需的类型进行传递。

1.8　边界对齐

边界对齐（boundary alignment）是处理器为了提高处理性能而对存取数据的起始地址所提出的一种要求。编译器为了使得我们所编写的 C 程序尽可能高效，就必须最大限度地满足处理器对边界对齐的要求。图 1.8 所示程序的运行结果是在终端上能看到“size of type_t is 8”。之所以为 8 而不是 5，正是因为编译器将 type_t 结构中的各变量进行了 4 字节边界对齐处理。

```
#include <stdio.h>

typedef struct {
    char a_;
    int b_;
} type_t;

int main ()
{
    printf ("size of type_t is %d\n", sizeof (type_t));
    return 0;
}
```

图 1.8

要对数据结构进行高效的操作，从处理器的角度来看，需要尽可能减少对内存的访问次数，尽管处理器包含了缓存（参见 1.11 节），但处理器在处理数据时还得读取缓存中的数据，显然，读取缓存的次数也越少越好。对于 32 位处理器，每一次读取操作都是 32 位的，即 4 个字节。图 1.9 示例说明了在大端模式的处理器上 type_t 结构的内存总局。图中的 b_变量包括 b_0、b_1、b_2 和 b_3 四个字节。

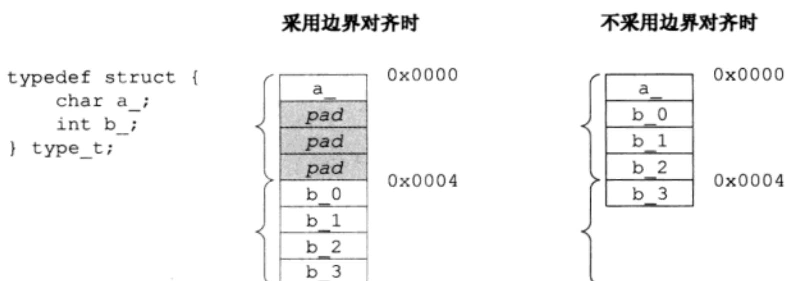


图 1.9[③]

在采用边界对齐处理的情形下，当处理器需要分别访问 a_和 b_变量时只需进行一次存取，图中的花括号表示一次存取操作。在不采用边界对齐的情形下，a_变量无论如何只要进行一次存取，而 b_变量却需要进行两次。更为麻烦的是，对于 b_还得将其合成一个 4 字节，这需要依靠更多的指令来完成，降低了程序的执行效率。

有些处理器在数据起始地址不满足边界对齐要求时，会引发异常（exception）而使得程序终止运行。图 1.10 是一段被简化的程序，分别在 Windows（基于 32 位的 x86 处理器）和 Solaris（基于 SPARC 处理器）上编译和运行其结果将完全不同。在 Windows 上程序能正常运行，但在 Solaris 上程序却会出错，并在终端上打印出“Bus Error”。造成这一现象的原因是：x86 处理器能自动地处理边界不对齐内存的访问，但是，SPARC 处理器却无法处理。

③ 图中的 pad 表示“填充”的意思，是因为边界对齐处理而留下的“空洞”。

```
typedef struct {
    short mark_;
    char body_ [128];
} msg_t;

typedef struct {
    char *pointer_;
} header_t;

int main ()
{
    msg_t msg = {0};
    void *p = ((header_t *)msg.body_)->pointer_;
    return 0;
}
```

图 1.10

下面我们分析一下为什么在 SPARC 处理器上程序会出错。在一个结构或联合体中，编译器会根据具体成员变量的类型选择边界对齐字节数，其选择依据是处理器的 ABI 规范（参见第 10 章）。对于 msg_t 结构，编译器将采用 2 字节对齐的处理方式；而对于 header_t 结构，则采用 4 字节对齐。通过 main.c 生成的汇编代码可以加以验证，如图 1.11 所示。

```
00010660 <main>:
int main ()
{
   10660:       9d e3 bf 00     save  %sp, -256, %sp
    msg_t msg = {0};
   10664:       82 07 bf 68     add  %fp, -152, %g1
   10668:       9a 10 20 82     mov  0x82, %o5
   1066c:       90 10 00 01     mov  %g1, %o0
   10670:       92 10 20 00     clr  %o1
   10674:       94 10 00 0d     mov  %o5, %o2
   10678:       40 00 40 55     call  207cc <memset@plt>
   1067c:       01 00 00 00     nop
    void *p = ((header_t *)msg.body_)->pointer_;
   10680:       c2 07 bf 6a     ld  [ %fp + -150 ], %g1
   10684:       c2 27 bf 64     st  %g1, [ %fp + -156 ]
    return 0;
   10688:       82 10 20 00     clr  %g1
}
   1068c:       b0 10 00 01     mov  %g1, %i0
   10690:       81 c7 e0 08     ret
   10694:       81 e8 00 00     restore
   10698:       81 c3 e0 08     retl
   1069c:       ae 03 c0 17     add  %o7, %l7, %l7
```

图 1.11

图中字体加粗的 ld 指令是从内存中读入一个 4 字节的字（word，这是处理器所定义的类型），其对应于 C 程序中取得 header_t 结构中的 pointer_变量的地址。在参考资料《The SPARC Architecture Manual *v8*》的 13 页有如下一段话需要特别注意。

其中的 halfword 是指 2 个字节，word 是指 4 个字节，而 doubleword 是指 8 个字节。这段话给我们的信息是：当使用 ld 指令从内存中读入一个 4 字节的字时，其地址必须是 4 字节边界对齐的。

Alignment Restrictions

Halfword accesses must be aligned on 2-byte boundaries, word accesses must be aligned on 4-byte boundaries, and doubleword accesses must be aligned on 8-byte boundaries. An improperly aligned address in a load or store instruction causes a trap to occur.

C 语言除了对结构或联合体内的变量进行对齐处理外（从结构内部的角度），还需要将整个数据结构分配在以 4 字节为边界的地方才有意义。对图 1.9 中所定义的 type_t 类型变量，即使结构体中的变量已经做过对齐处理，但如果整个结构不是放在 0x0000 地址（4 字节边界对齐）处，而是放在 0x0001 地址处，则整个结构中的变量都将变得边界不对齐。

回到图 1.10 中的 main.c 程序，我们可以分析出 main()函数中的 msg 变量的首地址是 4 字节边界对齐的，加上前面的大小为 2 个字节的 mark_后，msg.body_并不是 4 字节对齐的。接着程序将 msg.body 强制转换成了 header_t 结构。最终结果是 pointer_也是边界不对齐的，而这就违背了 SPARC 处理器中 ld 指令要求地址边界是 4 字节对齐这一限制。这也是为什么在 SPARC 处理器上运行这一程序时，会出现“Bus Error”的原因。

图 1.12 所示的代码却能在 SPARC 上正确地运行，这是因为编译器知道 body[1]所指的是 1 个字节，因此不会采用类似 ld 这样的指令去存取它，这可以从它的汇编代码看出，如图 1.13 所示。其中的 stb 指令表示向内存中写入一个字节，这一指令对于内存地址的边界对齐并无要求。

```
typedef struct {
   short mark_;
   char body_ [128];
} msg_t;

int main ()
{
   msg_t msg = {0};
   msg.body_[1] = 3;
   return 0;
}
```

图 1.12

```
00010660 <main>:
int main ()
{
  10660:       9d e3 bf 08     save  %sp, -248, %sp
   msg_t msg = {0};
  10664:       82 07 bf 68     add  %fp, -152, %g1
  10668:       9a 10 20 82     mov  0x82, %o5
  1066c:       90 10 00 01     mov  %g1, %o0
```

```
  10670:        92 10 20 00     clr  %o1
  10674:        94 10 00 0d     mov  %o5, %o2
  10678:        40 00 40 55     call  207cc <memset@plt>
  1067c:        01 00 00 00     nop
   msg.body_[1] = 3;
  10680:        82 10 20 03     mov  3, %g1     ! 3 <_START_-0xfffd>
  10684:        c2 2f bf 6b     stb  %g1, [ %fp + -149 ]
   return 0;
  10688:        82 10 20 00     clr  %g1
}
  1068c:        b0 10 00 01     mov  %g1, %i0
  10690:        81 c7 e0 08     ret
  10694:        81 e8 00 00     restore
  10698:        81 c3 e0 08     retl
  1069c:        ae 03 c0 17     add  %o7, %l7, %l7
```

图 1.13

在默认情形下，编译器将采用边界对齐的处理方法来提高程序的执行效率。但是，有时我们并不希望存在这种字节对齐处理。比如两台主机间进行网络通信时，我们并不希望因为字节对齐而传送多余的字节。为了避免编译器进行对齐处理，可以在结构之前加上“#pragma pack(1)”预处理指令，它告诉编译器对指定的数据结构采用单字节对齐的方式进行处理。图 1.14 是一段程序采用对齐方式和不采用对齐方式时汇编代码的比对。

图中 main0.c 是采用边界对齐的代码，从对应的汇编程序 main0.s 可以看出，只需要一个 st 指令（在 SPARC 处理器中，这一指令向内存中写入 4 个字节）即可。main1.c 是不采用边界对齐的代码，从其汇编程序 main1.s 中可以看出，其需要 2 个 ld 指令和 2 个 st 指令，这是因为在这种情况下变量 tp 跨越了 4 字节边界，需要通过拼凑的方式获得需要赋值的 b_变量，这与我们前面的分析是完全吻合的。从生成的代码长度来看，读者也不难想象采用边界对齐时程序的运行更高效。

```
typedef struct {
   char a_;
   int b_;
} type_t;

int main ()
{
   type_t tp;
   tp.b_ = 1;
   return 0;
}
```

```
0001063c <main>:
int main ()
{
  1063c:        9d e3 bf 88     save %sp, -120, %sp
   type_t tp;
   tp.b_ = 1;
  10640:        82 10 20 01     mov  1, %g1
  10644:        c2 27 bf ec     st  %g1, [ %fp + -20 ]
   return 0;
  10648:        82 10 20 00     clr  %g1
}
```

```
#pragma pack(1)
typedef struct {
    char a_;
    int b_;
} type_t;

int main ()
{
    type_t tp;
    tp.b_ = 1;
    return 0;
}
```

```
0001063c <main>:
int main ()
{
  1063c:       9d e3 bf 88      save %sp, -120, %sp
   type_t tp;
   tp.b_ = 1;
  10640:       fa 07 bf e8      ld  [ %fp + -24 ], %i5
  10644:       03 3f c0 00      sethi  %hi(0xff000000), %g1
  10648:       82 0f 40 01      and  %i5, %g1, %g1
  1064c:       c2 27 bf e8      st  %g1, [ %fp + -24 ]
  10650:       fa 07 bf ec      ld  [ %fp + -20 ], %i5
  10654:       03 00 3f ff      sethi  %hi(0xfffc00), %g1
  10658:       82 10 63 ff      or  %g1, 0x3ff, %g1     ! ffffff <_end+0xfdf74b>
  1065c:       82 0f 40 01      and  %i5, %g1, %g1
  10660:       3b 00 40 00      sethi  %hi(0x1000000), %i5
  10664:       82 10 40 1d      or  %g1, %i5, %g1
  10668:       c2 27 bf ec      st  %g1, [ %fp + -20 ]
   return 0;
  1066c:       82 10 20 00      clr  %g1
}
```

图 1.14

1.9　程序断点和数据断点

如果读者已有软件开发的经验，那一定知道什么是断点。通过断点我们可以方便地对程序进行调试。在嵌入式软件开发领域中，我们还得知道存在程序断点（program breakpoint）和数据断点（data breakpoint）之分。

程序断点就是指处理器的指令断点。通俗地说，就是当程序运行到某函数的某个地方时就会停下来。程序断点又可分为软件程序断点和硬件程序断点。

当用微软的 Visual Studio 进行软件调试时可以设置很多断点，这些断点都是软件程序断点。处理器在运行的过程中如果碰到了一条非法（或无效）的指令，就会出现一个异常中断，软件程序断点就是利用这个特性来实现的。当设置一个软件程序断点时，调试工具就在我们所想设置的内存位置上放置一条非法的指令，同时将被替换的指令保留起来。当程序运行到了被非法指令替换的地方时，处理器所产生的异常中断一方面在中断服务程序中恢复被替换的指令，另

一方面将控制权交给调试工具。从理论上说，软件程序断点可以设置 *n* 个，*n* 的大小由内存容量决定。

在嵌入式系统中，如果想调试的程序不是位于内存中，而是位于像闪存这样的存储器中（比如引导加载器的部分代码，参见第 19 章），此时就无法使用软件程序断点了，因为闪存中的内容并不能像内存那样方便更改。此时只能使用硬件程序断点来调试程序。硬件程序断点的实现原理与软件程序断点完全不同，断点是通过配置处理器的断点寄存器的方式实现的。当处理器运行到断点寄存器所指示位置的指令时就会产生中断，调试工具通过该中断使我们获得干预的机会。处理器所能设置的硬件程序断点数量是很有限的，可能最多也就 4 个。

程序断点清楚了，再看一看数据断点。当调试程序时，如果发现所定义的一个数据结构中的某一变量总是被意外地更改，查出这类问题的根源可并不容易。如果处理器能提供一种功能——当某一变量的值被更改时能自动停下来就好了，这样就可以通过调用栈找到问题的根源。这就是引入数据断点的目的。数据断点与硬件程序断点很相似，需要在处理器的寄存器中设置所监视数据变量的内存地址。当被监视的内存单元被修改时处理器将产生中断，调试工具利用这一中断让我们获得检查程序的机会。与硬件程序断点一样，数据断点的个数也很有限。

处理器一般都提供硬件程序断点这一功能，但数据断点却未必。选择处理器时考虑其是否支持数据断点是很有必要的，这会让我们获得另一种有效的调试手段。

1.10 内存管理单元

内存管理单元（Memory Management Unit，MMU）在现代处理器中扮演着非常重要的角色。操作系统通过使用处理器的内存管理单元，能实现以下功能。

- 虚拟内存。有了虚拟内存，可以在处理器上运行比实际物理内存大的应用程序。为了使用虚拟内存，操作系统通常要设置一个交换分区（通常是硬盘），通过将内存中不活跃的数据放入交换分区以腾出物理内存来为其他的程序服务。
- 内存保护。通过这一功能，可以将特定的内存块设置为读、写和可执行属性。

在嵌入式系统中通常不会使用虚拟内存这一功能，因为它会使操作系统的实时性更具不确定性。还有另一个原因就是，嵌入式系统的外部存储空间通常很小，且没有硬盘空间用做交换分区。

内存管理单元在嵌入式系统中主要用于实现内存保护。在 9.1 节中谈及程序的结构时提到了程序中的.text 段和.rdata 段，这两个段通常放在相邻的连续内存空间中，并通过内存管理单元实现只读和可执行保护。通过这一方法可以防止其内容被出错的程序意外地改写。对于设置成只读的内存区当被意外地改写时，处理器会产生一个异常中断，操作系统利用这一中断可以记录下出错时的函数调用栈，以帮助我们定位问题。

采用内存保护的方法有助于找到问题的根源，因为其更接近出错点；否则可能出现.text 段被意外更改后仍能运行，造成最终出错的地方离真正的出错点更远，这很不利于查错。还有，如果.text 段被更改了，那么程序的行为也可能会发生变化，这并不是我们所希望的。

内存管理单元的一个特性需要在此提及。内存管理单元中存在页的概念，对于所有与内存保护相关的操作都是以页的大小来进行的。比如，在常用的 32 位 x86 处理器上，页的大小是 4KB。在设置内存块的读写属性时，内存块的开始地址必须是页的整数倍，或者说地址必须是以页大小作为边界对齐的。

由于页的存在，在运用内存管理单元对.text 段进行保护时必须保证.text 段的起始地址是页对齐的，这可以通过链接器的脚本来达到目的（参见第 6 章）。同样由于内存管理单元是以页大小为保护单位的，因此得保证.text 段的大小也是页的整数倍，否则可能出现.data 段（它是可读写的）有一部分与.text 段的最后部分放入同一个页中。为了避免这一问题，将.data 段的起始地址也通过链接脚本设置成页对齐就行了。

这样看来，使用内存管理单元会浪费一点内存空间，因为当.text 段的大小不是页的整数倍时，.text 段与.data 段之间会存在空隙。

1.11　缓存

处理器在运行的过程中需要频繁地使用内存，以获取需要执行的指令和加工的数据。然而，处理器的运行速度远远高于外部内存的访问速度，如果处理器采用需要数据时就从内存中读取的方式，将大大降低处理器的性能。为了缓解外部内存访问这一瓶颈，大多数处理器都采用在其芯片内存中设计缓存的方式，这正是我们常听说的处理器的一级和二级缓存。由于缓存是位于处理器内部的，因此其工作频率可以很高，使得处理器对其存取不存在瓶颈。

当处理器需要从外部内存取数据时，先从缓存中查找数据是否以前存进来过。如果在缓存中找到了，则称为“缓存命中”，可将这一数据直接拿去使用。反之，如果没有找到，则需要从外部内存中进行读取。在这种情形下，并不是要一个字节就读一个字节，而是一次性地采用突发方式（burst mode，这是内存芯片支持的一种存取方式，这种方式比每次读取几个字节的方式更快）从内存中读入一行缓存行。一行缓存行可能是 64 个字节，这视不同的处理器而不同。

当处理器需要向内存写数据时，需要根据我们对缓存的配置进行不同的操作。其中一种模式是数据先写入缓存，然后由处理器在合适的时间将缓存中的数据回写到外部内存中；另一种模式是数据直接回写到外部内存中。

当处理器需要从外部内存读入数据时，可能需要在缓存中为将要读入的数据腾出空间来存放，此时处理器会根据哪些数据长时间没有使用这一准则找到最旧的缓存行，并用即将读入的数据覆盖它。在覆盖旧的缓存行之前，有可能需要先将它们回写到外部内存中。

缓存的设计是基于这样一种思想，即指令和数据的使用具有很强的局部性。比如一个函数是由多条指令组成的，而调用这一函数就意味着需要按序执行这些指令。如果在取函数的第一条指令时（为了说明方便，假设第一条指令位于缓存行的头部），通过一次性地快速读入多条指令的方式（大小为一行缓存行）来提高对外部内存的访问效率，那么通过后续的缓存命中就能提高存取效率。对于数据也存在同样的局部性特点，这一点相信读者不难理解。

在处理器中我们可以配置各地址空间是否需要使能缓存功能，或者说缓存并非只针对内存，还可以针对其他的存储器，如闪存。如果对应的空间是闪存，那么对闪存进行编程（又称为烧写）时一定要禁用缓存功能。这是因为在对闪存芯片进行编程时需要一定的写命令序列，如果闪存所在的空间使能了缓存功能，那么编程写命令序列有可能不会立即发送给闪存芯片，而是存放在缓存中，结果就是不能正常地进行闪存编程操作了。

在一些高端处理器上还提供了缓存锁定功能，使得某块缓存区中的数据不会因为腾出空间的需要而变成无效。这一功能对于那些需要频繁地查找数据表的程序很有用处。例如，可能有这样一张数据表，程序需要以高负荷的方式频繁地查找该表，将其锁定在缓存中能显著地提高处理器的执行效率。

1.12 小结

对于处理器基本概念的掌握不仅有助于开展嵌入式软件开发工作，还有助于我们更加深入地理解编程语言和更好地掌握计算机体系结构。



第 2 章

开发活动中的硬件问题

在出现的软件缺陷中，有些缺陷一发现就知道是软件设计上的错误，但有些缺陷无论从软件方面如何查找也发现不了问题。本章作者将与读者分享曾经经历的一些“灵异事件”，来帮助读者进一步了解嵌入式软件开发的复杂性。

2.1 两个案例

作者曾经参与一个 DVR（Digital Video Recorder，硬盘录像机）项目的嵌入式软件开发。在该产品的开发过程中出现过两个与硬件设计相关的问题。

第一个问题是，有时 DVR 在将图像数据写入硬盘时会造成死机，但从软件实现中却找不到问题所在。同样的程序放到另一块由 AMD 公司提供的参考设计板上进行测试时，结果显示程序很稳定。

第二个问题是，主板上的处理器有时无法收到键盘板的按键通知消息。通过各种调试手段，排除了键盘消息的丢失是因为按键失灵而造成的。在问题的排查过程中，硬件工程师建议将两板间两板间串口的通信速率（又称波特率）从 9600 降到 4800，当波特率降到 4800 后问题得到了解决。

讲述这两个问题是想告诉读者，嵌入式软件的开发与硬件息息相关。对于桌面和服务器软件的开发，其硬件平台是由大型厂商生产的，这些大厂商不光硬件设计能力强，测试也做得充分，且用户数量也大（这容易暴露设计缺陷），因此硬件平台的稳定性通常不需要软件工程师关心。然而，大多的嵌入式产品其处理器板卡都是公司自行设计的（包括原理图和印制板），容易因为开发资源的不足而出现硬件质量问题。其中最为常见和严重的问题是印制板的信号完整性问题，上面所说的两个问题正是由它所导致的。

2.2 案例的背后——信号完整性

硬件设计大致分为两个阶段。第一个阶段就是硬件原理设计，这一过程的输出结果就是原理图。有了原理图以后，第二个阶段就是设计印制板。原理图中的错误相对容易被发现，因为原理行不通会直接影响产品的功能。外行人看来，印制板的设计很简单，“只要原理图没有问

题，连连线路板上的线就可以了”，实则不然。

在数字电路中，数据是通过信号线上电平的高低变化来传递的，电平的高低分别代表程序中的 1 和 0。理论上，电平的高低变化在线路上应当产生像图 2.1（a）中那样的波形，即菱角分明的方波。但是，由于线路中电容效应的存在，使得所获得的波形可能如图 2.1（b）中的那样，即波形的上升沿与下降沿并不陡峭，而信号的陡峭程度与印制板的设计（指布线）有关。信号的完整性就是指信号实际波形与理想波形的接近程度，越是接近理想波形则信号的完整性就越好。

当数据传递的速度越快时，所表现出来的就是在信号线上，电平的高低变化的频率也越快，即波形的宽度也更窄。图 2.1（c）示例说明了频率更快时的畸变波形。图 2.2 很好地示例说明了一根线上的信号从传输端到接收端的波形变化。图中的信号是从处理器向外设芯片传输的，处理器输出信号时是一个方波，但经过印制板上的线路传输后，到达外设芯片时其波形却发生了畸变。当畸变足够严重时，外设芯片就很有可能不能正常识别出处理器所发出的信号。

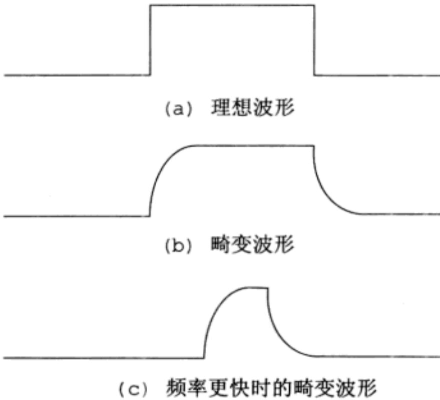


图 2.1

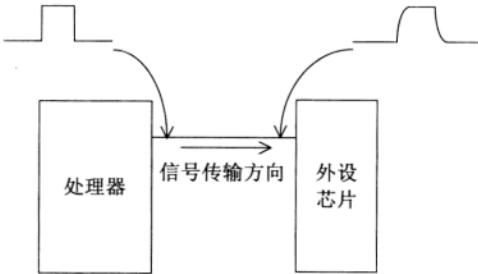


图 2.2

由于信号完整性问题的存在，所以设计印制板并没有想象的那么简单。当所设计的电路属于低频电路时，信号完整性问题并不严重，甚至可以忽视；但当所设计的是高频电路时，信号完整性问题就变得突出了。

了解信号完整性问题产生的机理需要运用到一定的数学知识——傅立叶级数变换。理论上，一个方波可以通过一定频率的正弦波（包括高次谐波）叠加而成。由于传输线中的电抗对不同次数的谐波有不同程度的影响，结果会造成不同的谐波经过同样的传输线传送以后，产生不同的相位差和幅度衰减。最终，当这些波到达了接收端时，其叠加起来的结果就不再是一个方波了。对于完整性问题更为深入的知识，读者可以在 Internet 上搜索一下。

印制板的设计将决定信号的完整性，其中最为重要的就是印制板上元件的布局和布线方法。如果读者曾经注意观察过计算机主板，在主板上能看到“蛇”形的走线，这种布线方式正

是为了消除信号完整性问题的。因为处理器与外部芯片间的工作频率很高时，信号线长度的不同会造成不同的相位差而导致信号完整性问题。

前面所说的对于向硬盘写数据所出现的死机现象，当将同样的程序放到 AMD 所提供的参考板上运行时问题就消失了，这正是因为 AMD 参考板上的电路布线设计得更好。硬件工程师后来解决问题的方法，就是重新对硬盘接口和处理器之间的连线进行重新布置。同样地，所出现的键盘按键消息丢失的现象也是因为布线而造成的，因为其中一条传输线会受到其周边元件的干扰而出现信号完整性问题。

2.3 应对方法

在软件开发过程中如何应对信号完整性这类问题呢？下面给出几点建议。

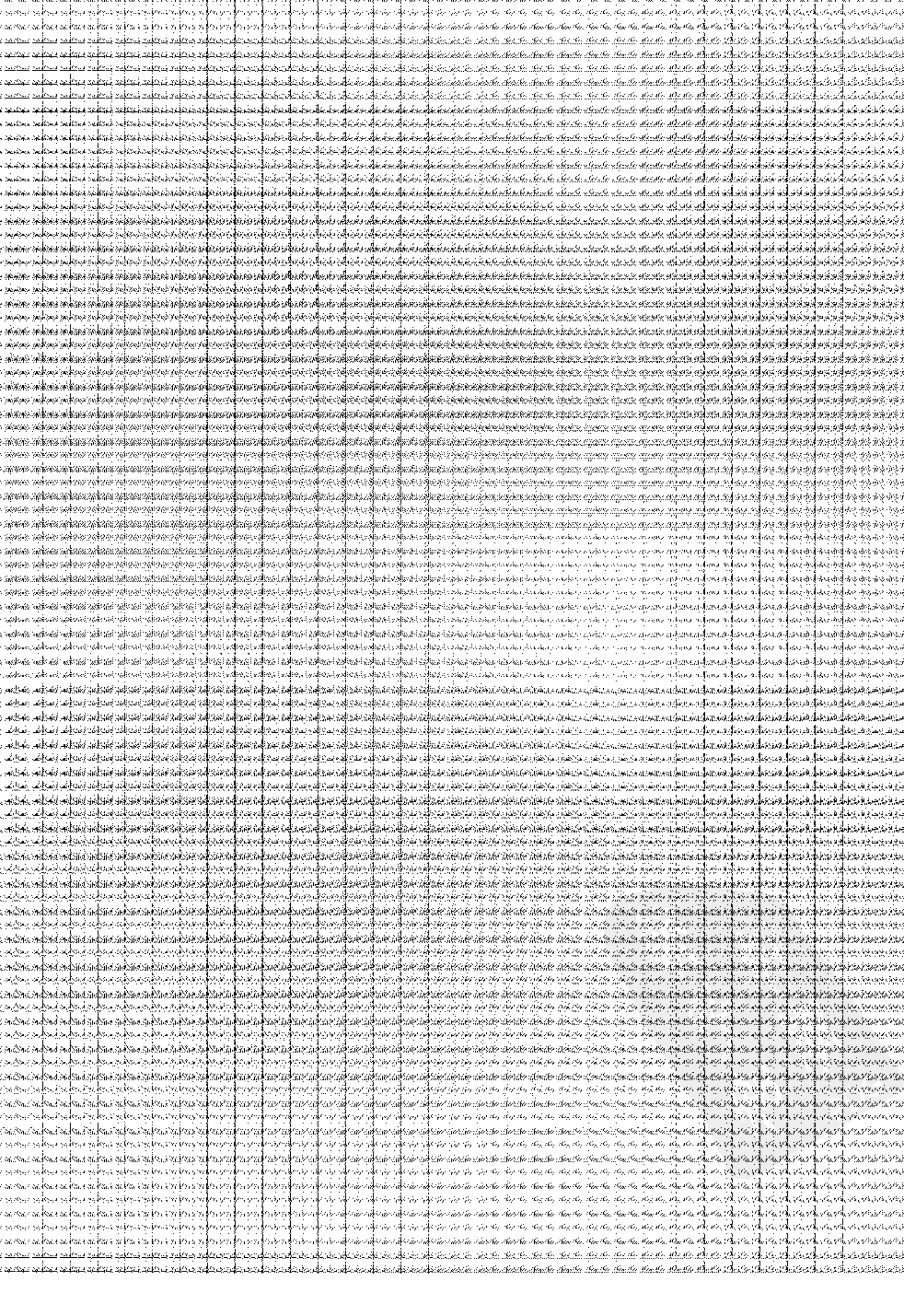
- 在怀疑信号完整性问题之前要确保软件没有错误。
- 在开发过程中，最好在手边有一块大公司所提供的参考开发板卡。这里的隐含假设是大公司的参考板布线质量有保障。参考板的存在能有效地界定所出现的问题是由硬件还是软件引起的。
- 从前面的分析可以看出，处理器与外设芯片之间信号的工作频率越快就越容易出现信号完整性问题。为此，对于处理器与外设的信号线（地址总线、数据总线、片选信号等）其工作频率不应一味地追求快，而应够用就行了。这一点，可以通过给处理器的片选空间配置更为宽松的时序做到。

2.4 小结

在嵌入式系统中，软件质量与硬件质量是息息相关的。硬件质量不只取决于正确的硬件原理，还与印制板的设计有关。不良的印制板设计很容易导致信号完整性问题，并进一步引发软件莫名其妙的错误。

当读者在嵌入式软件开发的过程中发现软件出现莫名其妙的错误时，应大胆怀疑是硬件不稳定所导致的，但小心求证。





工具篇

软件开发离不开各种工具的支撑，嵌入式软件开发更是如此。在嵌入式软件开发中，如果不精通一些工具，那很难体现我们的专业，也谈不上高效工作。由于嵌入式软件开发中 GNU 工具的使用占有相当大的比重，因此本篇将以 GNU 工具为例来介绍相关工具。

与使用 Visual Studio 这样出色的图形化集成开发环境进行软件开发不同的是，嵌入式软件开发大多是基于命令行的。当然，也存在像来自 WindRiver 的 Workbench 这样的集成嵌入式开发环境，但使用它的人毕竟是少数。如果是使用命令行的开发环境，那就面临如何有效地设计开发环境的问题，其中编译环境的设计首当其冲。开发环境的设计需要我们能完全驾驭 make，因为它是开发环境方面的全能管家。第 3 章将对控制 make 行为的 Makefile 进行细致的介绍。一旦掌握该章的内容，读者将达到 Makefile 方面的“准专家”级水平。想成为 Makefile 方面的真正专家，离不开读者的进一步实践。

在程序开发的过程中需要通过编译器生成可执行程序。编译器对于大多的工程师来说只是一个命令加上一大堆选项，孰不知其中有些选项能极大地提高我们的工作效率和帮助探究更深层次的技术知识。第 4 章我们将一同回顾 gcc 编译器的幕后行为，以及那些值得掌握和加以运用的选项。

一旦生成了可执行程序，就可以加载到嵌入式设备上运行，至于程序文件中到底包含些什么我们通常不用关心吧？不！对于嵌入式软件开发工程师来说，需要清楚地了解程序文件的细节。对于程序文件内部信息的了解需要通过工具，这是引入第 5 章介绍 binutils 工具集的原因。毫不夸张地说，binutils 是嵌入式软件开发的一个“利器箱”，掌握它们使得我们在日常工作时轻松很多。

编写和阅读链接器的链接脚本是嵌入式软件工程师需要掌握的技能之一。这对于理解处理器上电时的启动代码是如何被放置在特定的存储空间是有益的。在第 6 章我们将一同探讨 ld 链接器链接脚本的语法和它的几个常用选项。

嵌入式软件开发工作离不开对所编写的代码进行调试，这需要用到像 gdb 这样的调试工具。尽管存在基于 gdb 的图形化调式工具，但掌握以命令行方式使用 gdb 的技能仍然是我们的学习目标，因为在一些情况下除了这一方式没有别的选择。第 7 章用于帮助读者掌握在命令行方式下调试软件。

一个工具要用好，不能只停留在会用上，而应努力做到用精。因为用精意味着更高效，也是我们专业化的需要。让我们别忘了“工欲善其事，必先利其器”这句古话。

第 3 章 make，开发环境全能管家

高效的开发环境能显著地提高开发效率。在嵌入式软件开发领域，大量的开发环境是通过使用 make 工具来构建的。而要使用 make 工具，就离不开编写 Makefile。

本章将从使用 make 实现项目程序编译的角度，来介绍如何编写 Makefile 文件去“指挥”make 为我们编译程序。在本书中，make 和 Makefile 是两个可互换的术语。本章最终所构建的编译环境与“专业”还有一定的距离，在本书最后的质量保证篇中我们还将延续开发环境构建这一话题，介绍如何通过使用 make 搭建一个专业和高效的开发环境（注意，不只是编译环境）。

当编译一个程序时，与编译相关的问题主要有两方面：一方面是编译器报告的程序代码中的语法错误；另一方面就是与 Makefile 相关的错误。从事嵌入式软件开发如果不能驾驭 Makefile，那就很难做到游刃有余。

项目编译是我们的开发工作中周而复始、非常频繁的动作。如何构建一个高效的项目编译系统，是我们必须去思考的。下面将从最简单的“Hello World”开始，到本章最复杂的 huge 虚拟项目，逐步深入地介绍使用 Makefile 构建高效编译系统所需掌握的知识。

3.1 从最简单的 Makefile 中了解规则

学习 Makefile，最重要的是要掌握 3 个概念，分别是目标（target）、依赖关系（dependency）和命令（command）。目标就是指要干什么，或者说运行 make 后生成什么；依赖是指明目标所依赖的其他目标；命令则告诉 make 如何生成目标。这 3 个概念是通过 Makefile 中的规则（rule）关联在一起的。

对于一个真实的软件项目，为了通过 make 生成最终的目标，需要产生大量的中间目标，而各规则所描述的依赖关系就是将所有的目标关联在一起，结果就是使得我们可以只运行一次 make 命令，而链式反应地创建整个软件项目的所有目标——编译每一个源文件生成目标文件、生成库，以及创建可执行文件。是否真正驾驭 Makefile 的标志，就看能否运用目标和依赖关系去思考和表达需要 make 为我们所做的事情。

“Hello World”示例程序是很多编程语言入门时所讲解的第一个程序。同样地，我们也可

以编写一个能在终端上输出“Hello World”的简单 Makefile 以扬起学习之帆。

使用读者所熟悉的文本编辑器，编写内容如图 3.1 所示的 Makefile 文件，文件的存放目录可以是任意的。

```
all:
	echo "Hello World"
```

图 3.1

Makefile 中的第一个重要概念是目标（target）。在图 3.1 所示的 Makefile 中，all 就是一个目标，目标名是放在“:”前面的，名字可以由字母和下划线组成。这里的 all 目标是一个抽象的概念，在此应将其理解为“在终端上打印‘Hello World’”这一行为。

“echo "Hello World"”就是生成目标的命令。生成目标的命令可以是操作系统命令行中的命令或 make 所定义的函数。在 Linux 操作系统中，echo 命令的用处是将字符串打印到终端上，它与 C 语言中的 printf()函数的功能很相似。

请注意，命令所在的行必须是以 Tab 键开头。很多初学者容易犯的“低级”错误是，用空格代替开头的 Tab 键。

在图 3.1 的 Makefile 中，目标和命令组合在一起就形成了一个简单的规则。通过这个规则，我们告诉 make 要做什么。下面看一看这个 Makefile 的运行结果，图 3.2 示例说明了三种不同的运行方式及所获得的相应结果。

```
                    /Makefile/helloworld
$ make
echo "Hello World"
Hello World
                    /Makefile/helloworld
$ make all
echo "Hello World"
Hello World
                    /Makefile/helloworld
$ make test
make: *** No rule to make target 'test'.  Stop.
```

图 3.2

第一种运行方式是在 Makefile 所在的目录下运行 make 命令且不带任何参数。可以看到，最终会在终端上输出两行，第一行是 make 打印出来的将要运行的命令，这一命令也是我们写在 Makefile 中的，第二行则是命令的运行结果。

第二种运行方式是运行 make 命令并指定 all 参数。all 参数是告诉 make 我们希望构建 all 目标。一个 Makefile 中可以定义多个目标，在运行 make 命令时可以指定具体的目标加以选择。从运行结果来看，这种运行方式的效果与第一种运行方式是一样的。第一种运行方式虽然没有指定目标，但却获得了与指定 all 目标相同的运行结果，这是因为存在默认目标的缘故。关于

默认目标，后面还会提及。

第三种运行方式则是运行 make 并带上 test 参数，告诉 make 我们希望生成 test 目标。由于在 Makefile 中根本没有定义 test 目标，所以运行结果是可想而知的。make 的确报告了不能找到用于构建 test 目标的规则。

现在，让我们对图 3.1 的 Makefile 做一点小小的改动，如图 3.3 所示。其中的改动就是增加了 test 规则用于构建 test 目标——在终端上打印出“Just for test!”。图 3.4 是修改后的运行结果。

```
helloworld/Makefile
all:
    echo "Hello World"
test:
    echo "Just for test!"
```

图 3.3

```
                    /Makefile/helloworld
$ make
echo "Hello World"
Hello World
                    /Makefile/helloworld
$ make test
echo "Just for test!"
Just for test!
```

图 3.4

从这两个版本的 Makefile 的运行结果中，我们学到了如下几点。

- 一个 Makefile 中可以定义多个目标。
- 调用 make 命令时，得告诉它我们希望它构建的目标是什么，即要它干什么。当没有指明具体的目标时，make 将以文件中定义的第一个目标作为这次运行的目标。“第一个”目标，也被称为默认目标。
- 当 make 得到目标后，先找到构建目标的对应规则，然后运行规则中的命令来达到构建目标的目的。目前的 Makefile 中每个规则中都只有一条命令，实际上，一个规则中可以根据需要存在多条命令。

从目前的运行结果来看，当运行 make 时，make 会打印出在 Makefile 中即将被运行的每一条命令。大多情形下，这种运行方式会让人觉得很冗余，甚至混乱。要使 make 不输出它们，只要做一点小小的修改就行了。改过的 Makefile 在命令前加了一个“@”，如图 3.5 所示。更改后的运行结果位于图 3.6 中。

```
all:
    @echo "Hello World"
```

图 3.5

```
                    /Makefile/helloworld
$ make
Hello World
```

图 3.6

再对图 3.3 中的 Makefile 做一点改动，如图 3.7 所示。其中的改动之一是在各命令前增加了“@”，之二则是在 all 目标之后加上了 test 目标。对修改后的 Makefile，采用不同参数的运行结果列于图 3.8 中。

```
all: test
    @echo "Hello World"
test:
    @echo "Just for test!"
```

图 3.7

```
                    /Makefile/helloworld
$ make
Just for test!
Hello World
                    /Makefile/helloworld
$ make test
Just for test!
```

图 3.8

从输出结果可以发现，当不带参数运行 make 时，构建 all 目标之前 test 目标也被构建了。现在需要引入 Makefile 中的依赖关系这一概念。图 3.7 中 all 目标后面的 test 是告诉 make，all 目标依赖于 test 目标，这个依赖目标又被称为（all 目标的）先决条件（prerequisite）。

出现这种目标依赖关系时，make 会按从左到右（指在同一规则中的）和从上到下（指在不同规则中的）的先后顺序先构建一个规则所依赖的每一个目标，形成一种“链式反应”。对于这里的 Makefile，在构建 all 目标之前 make 会先构建 test 目标，这也体现了为什么一个规则中的依赖目标被称为先决条件的原因。图 3.9 采用 UML 的类图表达了两个目标之间的依赖关系。

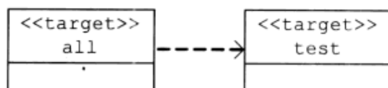


图 3.9

至此，读者已经认识了 Makefile 中的细胞——规则，图 3.10 是规则语法的 UML 表示，而图 3.11 是其文字描述形式。

一个规则是由目标、先决条件以及命令组成的。需要指出的是，目标和先决条件之间表达的就是依赖关系（dependency），这种依赖关系指明在构建目标之前，必须保证先决条件先满足（即先被构建）。

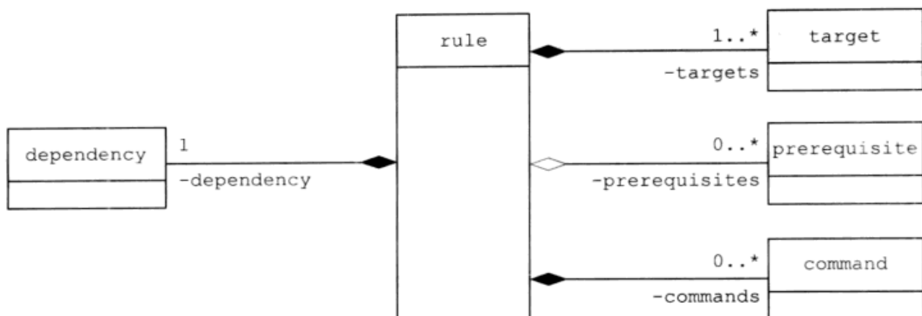


图 3.10

```
targets : prerequisites
    command
    ……
```

图 3.11

一个规则可以定义多个目标，当一个规则中存在多个目标且这个规则是 Makefile 中的第一个规则时，如果运行 make 命令不带任何目标，那么规则中的第一个目标将被视为默认目标。图 3.12 的 Makefile 和图 3.13 对应的运行结果可以帮助理解这一点。

```
all test:
    @echo "Hello World"
```

图 3.12

```
$ make
Hello World
$ make test
Hello World
```

图 3.13

Makefile 说起来也很简单，因为其基本单元就是规则，不论多么复杂的 Makefile 都是用规则“码”出来的。当然，为了更高效地“码”出来，还得运用 Makefile 所提供的其他语法。

规则的功能就是指明 make 在什么时候，以及如何来为我们（重新）构建目标。在“Hello World”这个例子中，不论在什么时候运行 make 命令（带目标或者不带目标），其都会在终端上打印出信息（即有真正的动作发生），这一点和我们采用 make 进行程序编译时的表现有所不同。另外，采用 Makefile 进行代码编译时，Makefile 中所存在的先决条件大多是指具体的程序文件。

make 处理一个规则的活动图如图 3.14 所示，当中的“构建先决条件”就是重复图 3.14 所示的同样的活动，读者可以将其看做是对图 3.14 所示的活动图的递归调用。而“运行命令构建目标”是由命令组成的一系列动作。

为了更深刻地理解图 3.14 所示的活动图，让我们以图 3.7 中的 Makefile 为例，看一看 make 是如何做出响应的。具体化后的活动图如图 3.15 所示。图中的上部分是 all 规则的处理活动图，由于 all 规则有一个 test 依赖目标，所以其走的是“存在依赖关系”分支进行“构建‘test’目录”，最后，运行 all 规则的 echo 命令。图中的下部分则是构建 test 目标的活动图，由于 test 目标没有依赖关系，所以走的是“否则”分支。

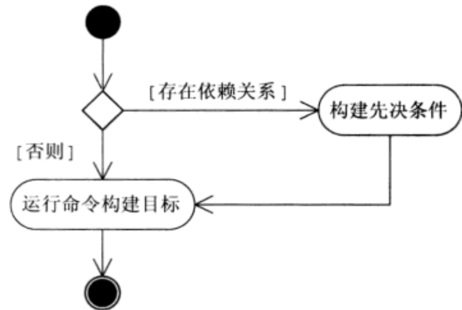
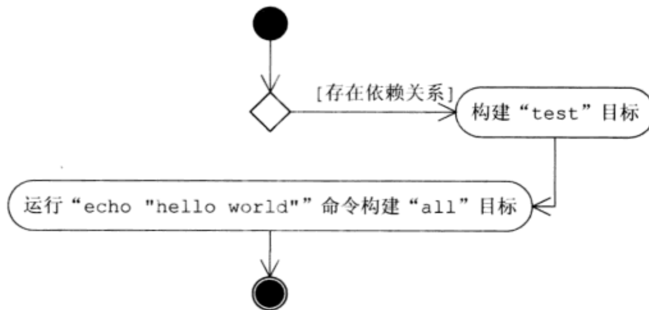
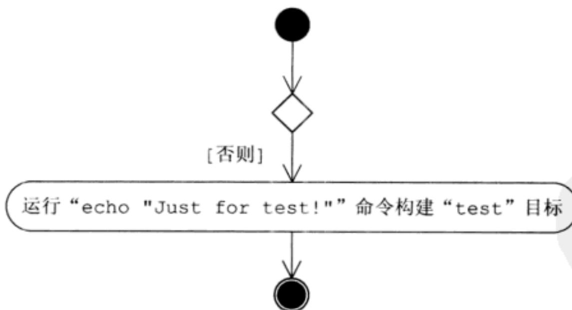


图 3.14

虽然通过“Hello World”例子我们只认识了 Makefile 中的规则，但这是很不错的一个开端。后面将以做虚拟项目的形式，从最简单的 simple 项目到本章最复杂但却实用的 huge 项目，逐步深入介绍 Makefile 中的知识点。



构建“all”目标的活动



构建“test”目标的活动

图 3.15

3.2　创建基本的编译环境

下面让我们从 simple 这个虚构的简单项目开始，尝试着将规则运用到项目的编译系统中。

3.2.1 将规则运用于程序编译

先假设有图 3.16 所示的用于创建 simple 项目可执行程序的两个源程序文件。如何为之创建编译所需的 Makefile 呢？

```
#include <stdio.h>

void foo ()
{
   printf ("This is foo ()!\n");
}
```

```
extern void foo ();

int main ()
{
   foo ();
   return 0;
}
```

图 3.16

编写 Makefile 的第一步，不是一个猛子扎进去试着写一个规则并对之调试，而应先采用面向依赖关系的思考方法勾勒出 Makefile 要表达怎样的依赖关系，这一点至关重要。通过不断地练习这种思考方法，才可能达到流畅地编写 Makefile。让我们看看 simple 项目的依赖关系应是怎样的。

图 3.17 或许是第一个跃入我们脑海的依赖关系类图，它表示 simple 可执行文件是通过 main.c 和 foo.c 编译生成的。通过这个依赖关系图就可以写出一个 Makefile 来了，但这个任务交给读者来完成。之所以这里不讲，是因为基于这样的依赖关系所写出来的 Makefile 在现实项目中的可维护性很差。

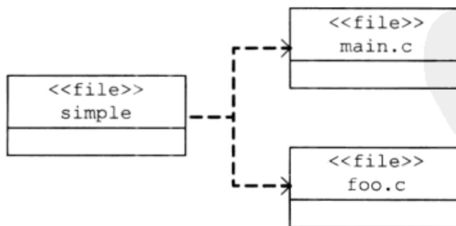


图 3.17

那怎样的依赖关系能使我们写出更具维护性的 Makefile 呢？图 3.18 实现了对依赖关系的更精确的表达，其中可以看到目标文件的身影。通过增加源程序文件所对应的目标文件，将有助于写出表达能力更强的 Makefile，这也意味着所获得的 Makefile 更具可维护性。

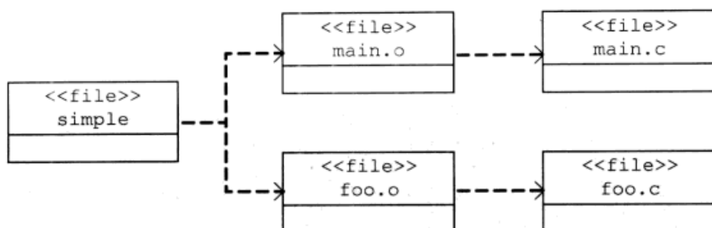


图 3.18

对于 simple 可执行程序来说，图 3.18 表示的就是它的“依赖树”。有了“依赖树”编写 Makefile 就会相对轻松，接下来要做的就是将其中的每一个依赖关系用 Makefile 中的规则进行描述。图 3.19 是所对应的 Makefile，而图 3.20 展示了依赖关系与规则间的映射。

```
all: main.o foo.o
   gcc -o simple main.o foo.o
main.o : main.c
   gcc -o main.o -c main.c
foo.o: foo.c
   gcc -o foo.o -c foo.c
clean:
   rm simple.exe main.o foo.o
```

图 3.19

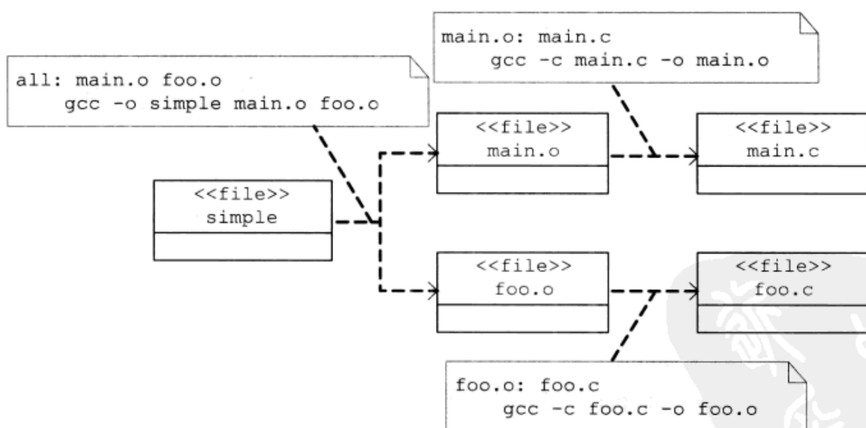


图 3.20

Makefile 中增加了一个 clean 目标用于删除编译所生成的文件，包括目标文件和 simple 可执行程序。读者可能注意到了，all 规则中生成的是“simple”，但 clean 规则为什么是“simple.exe”呢？这里有个知识点需要掌握，在 Cygwin 环境中 gcc 生成可执行文件时仍然遵照 Windows 操作系统的规则，会增加一个.exe 后缀。即使我们指定的生成文件名是“simple”，但 gcc 实际生成的却是“simple.exe”。

图 3.21 给出了 simple 项目的编译、执行以及清除的运行结果。从结果来看，我们已小有所成。

```
                    /Makefile/simple
$ make
gcc -c main.c -o main.o
gcc -c foo.c -o foo.o
gcc -o simple main.o foo.o
                    /Makefile/simple
$ ./simple.exe
This is foo ()!
                    /Makefile/simple
$ make clean
rm simple.exe main.o foo.o
```

图 3.21

在不修改源程序文件的清况下再执行一次 make，会出现什么现象呢？图 3.22 给出了答案。

```
                    /Makefile/simple
$ make
gcc -o simple main.o foo.o
```

图 3.22

第二次编译并没有构建目标文件的动作，但有构建 simple 可执行程序的动作。为了明白为什么，我们需要了解 make 在每一次构建时，是如何决定哪些目标需要重新创建的。

make 是通过文件的时间戳来判定哪些文件需要重新编译。通过前面已经提到的目标和先决条件之间的依赖关系，make 在分析一个规则以创建目标时，如果发现先决条件中文件的时间戳大于目标的时间戳，即先决条件中的文件比目标更新，就知道需要运行规则当中的命令重新构建目标。这意味着 make 在编译项目时对于系统上的时间有一定的要求，如果人为地变更系统时间或者将一个已编译好的项目拷贝到另一个主机上时，会造成 make 不能正常工作。碰到这种问题时，可以尝试通过执行一次“make clean”去解决。如果还解决不了问题，可以参照后面 3.8 节所介绍的另一个终极方法。

知道了 make 是如何工作的后，我们不难想明白为什么进行第二次 make 时还会重新构建 simple 可执行文件，因为 simple 文件并不存在。前面已指出在 Cygwin 上生成的可执行文件名是“simple.exe”，而不是“simple”。所以为了避免这个不一致带来的 simple 重新构建的问题，我们需要修改 Makefile，修改后的 Makefile 如图 3.23 所示。

```
all: main.o foo.o
   gcc -o simple.exe main.o foo.o
main.o : main.c
   gcc -o main.o -c main.c
foo.o: foo.c
   gcc -o foo.o -c foo.c
clean:
   rm simple.exe main.o foo.o
```

图 3.23

更改后的运行结果如图 3.24 所示，结果与前面是一样的。这是因为 Makefile 中的第一条规则中的目标是 all，而 all 文件在编译过程中并不生成，即 make 在第二次编译时找不到它，所以又重新构建 all 目标，而这导致了 simple.exe 被再一次生成。

```
                 /Makefile/simple
$ make
gcc -o simple.exe main.o foo.o
```

图 3.24

再一次修改后的 Makefile 如图 3.25 所示。其中将 all 目标变成了 simple.exe。

```
simple/Makefile
allsimple.exe: main.o foo.o
    gcc -o simple.exe main.o foo.o
main.o : main.c
    gcc -o main.o -c main.c
foo.o: foo.c
    gcc -o foo.o -c foo.c
clean:
    rm simple.exe main.o foo.o
```

图 3.25

图 3.26 是新的运行结果。这次 make 的确发现了不需要对 simple.exe 进行重新构建。make 的这一行为正是我们所希望的。

```
                 /Makefile/simple
$ make
make: `simple.exe' is up to date.
```

图 3.26

下面来验证一下如果对 foo.c 进行改动，是不是 make 能正确地发现并重新构建 foo.o 乃至最终的 simple.exe 文件。从 make 的角度来看，一个文件是否改动不是看文件大小和内容，而是其时间戳。通过使用 touch 命令可以改变文件的时间戳，这相当于模拟对文件进行了一次修改。图 3.27 列出了验证所需的所有步骤。

```
                 /Makefile/simple
$ ls -l foo.c
-rw-r-r-1 Administrator None 70 Aug 14 07:38 foo.c
                 /Makefile/simple
$ touch foo.c
                 /Makefile/simple
$ ls -l foo.c
-rw-r-r-1 Administrator None 70 Aug 14 08:48 foo.c
                 /Makefile/simple
$ make
gcc -c foo.c -o foo.o
gcc -o simple.exe main.o foo.o
```

图 3.27

从验证结果来看，make 发现了 foo.c 需要重新编译，也适时地对 simple.exe 进行了重新构建。

3.2.2 让 Makefile 更专业

现有的 Makefile 虽然能工作，但不够灵活。编写出一个专业、高效的 Makefile，还需要我们掌握更多的其他知识。第一个需要了解的是假目标。

3.2.2.1 假目标的用处

在 simple 项目中，假设 Makefile 所在目录下存在一个 clean 文件（可以通过 touch 命令来创建）。如果此时运行“make clean”，读者将发现 make 总是提示 clean 文件是最新的，而不是按我们所期望的那样对项目进行文件清除操作，如图 3.28 所示。

```
                        /Makefile/simple
$ touch clean
                        /Makefile/simple
$ make clean
make: `clean' is up to date.
```

图 3.28

make 的这种行为从原理上还是可以理解的，因为它将 clean 当做文件来处理。由于在当前目录下找到了这个文件，加上 clean 目标没有任何先决条件，所以当要求 make 为我们构建 clean 目标时它就会认为 clean 文件是最新的，从而“拒绝”进行真正的文件清除操作。

出现这种情形，是因为我们对于 clean 目标的定义与 make 所理解的有所出入。目录文件名与 Makefile 中的目标名重名在现实项目中是难免的，假目标（phony target）概念的提出正是为了解决这种问题的。

假目标采用.PHONY 关键字来定义，注意它必须是大写字母。图 3.29 是将 clean 变为假目标后的 Makefile。更改后运行“make clean”的结果如图 3.30 所示。

```
.PHONY: clean
simple.exe: main.o foo.o
    gcc -o simple.exe main.o foo.o
main.o : main.c
    gcc -o main.o -c main.c
foo.o: foo.c
    gcc -o foo.o -c foo.c
clean:
    rm -fr simple.exe main.o foo.o
```

图 3.29

```
                        /Makefile/simple
$ make clean
rm -fr simple.exe main.o foo.o
```

图 3.30

采用.PHONY 关键字声明一个目标后，make 并不会将其当做一个文件来处理。可以想象，由于假目标并不与文件关联，所以每次构建假目标时它所在规则中的命令一定会被执

行。拿这里的 clean 目标做比方，即使连续运行多次“make clean”，make 每次都会进行文件清除操作。

3.2.2.2　运用“变量”提高可维护性

编写专业的Makefile同样离不开运用变量，通过使用变量可以使得Makefile更具可维护性。让我们看一看如何通过使用变量来提高 simple 项目 Makefile 的可维护性，图 3.31 是运用变量的第一个 Makefile。

```
.PHONY : clean

CC = gcc
RM = rm

EXE = simple.exe
OBJS = main.o foo.o

$(EXE) : $(OBJS)
	$(CC) -o $(EXE) $(OBJS)
main.o : main.c
	$(CC) -o main.o -c main.c
foo.o: foo.c
	$(CC) -o foo.o -c foo.c
clean :
	$(RM) -fr $(EXE) $(OBJS)
```

图 3.31

这次我们定义了 CC、RM、EXE、OBJS 四个变量。定义变量时其值可以为空，即无右值。引用变量需要采用“$（变量名）”或“${变量名}”的形式。

引入变量的好处很明显，比如引入 CC 变量以后，如果需要更改编译器，只需更改变量赋值这一个点即可。Makefile 中变量的数据类型，可以理解为 C 语言中的字符串。

1. 自动变量

图 3.31 的 Makefile 中，存在目标名和先决条件名在规则的命令中重复出现。如果目标名或先决条件名发生了改变，那得在相应的命令中跟着改，这很麻烦。为了省去这种麻烦，我们可以借助如下一些自动变量。

- $@：用于表示一个规则中的目标。当一个规则中有多个目标时，$@所指的是其中任何造成规则命令被运行的目标。
- $^：表示的是规则中的所有先决条件。
- $<：表示的是规则中的第一个先决条件。

除了这三个自动变量外，在 Makefile 中还可以使用其他的自动变量，后面在需要用到时会提及。就 simple 项目的 Makefile 而言，为了简化它，采用这三个变量就足够了。图 3.32 是用于测试这三个自动变量的 Makefile，运行结果如图 3.33 所示。

```
Makefile
.PHONY: all

all: first second third
    @echo "\$$@ = $@"
    @echo "$$^ = $^"
    @echo "$$< = $<"

first second third:
```

图 3.32

```
$ make
$@ = all
$^ = first second third
$< = first
```

图 3.33

上例中还有几个地方需要注意。第一，在 Makefile 中“$”具有特殊的意思，如果想采用 echo 输出“$”，则必须用两个连着的“$”；第二，“$@”对于 Bash Shell 也有特殊的意思，需要在“$$@”之前再加一个脱字符“\”（引号不包含在内）。图 3.32 的最后一行是一个只有目标的规则，如果去除它会出现什么问题呢？读者自己可以试试看。

采用自动变量后，simple 项目的 Makefile 可以被重写为图 3.34 那样。

```
simple/Makefile
.PHONY: clean

CC = gcc
RM = rm

EXE = simple.exe
OBJS = main.o foo.o

$(EXE) : $(OBJS)
    $(CC) -o $@ $^
main.o : main.c
    $(CC) -o $@ -c $^
foo.o: foo.c
    $(CC) -o $@ -c $^
clean:
    $(RM) -fr $(EXE) $(OBJS)
```

图 3.34

2．特殊变量

在 Makefile 中，有两个特殊变量会经常用到：MAKE 和 MAKECMDGOALS。MAKE 变量表示的是当前处理 Makefile 的命令名是什么。在本书的例子中，$（MAKE）的值就是“make”。当需要在 Makefile 中运行另一个 Makefile 时，需要用到这个变量。图 3.35 是对 MAKE 变量进行验证的 Makefile，图 3.36 是验证结果。

```
.PHONY: all

all:
	@echo "MAKE = $(MAKE)"
```

图 3.35

```
$ make
MAKE = make
```

图 3.36

MAKECMDGOALS 变量表示的是当前构建的目标名。图 3.37 和图 3.38 分别是测试它的 Makefile 和测试结果。

```
.PHONY: all clean

all clean:
	@echo "\$$@ = $@"
	@echo "MAKECMDGOALS = $(MAKECMDGOALS)"
```

图 3.37

```
$ make
$@ = all
MAKECMDGOALS =

$ make all
$@ = all
MAKECMDGOALS = all

$ make clean
$@ = clean
MAKECMDGOALS = clean

$ make all clean
$@ = all
MAKECMDGOALS = all clean
$@ = clean
MAKECMDGOALS = all clean
```

图 3.38

从测试结果来看，MAKECMDGOALS 变量指的是用户输入的目标，当只运行 make 命令且不带参数时，虽然根据 Makefile 的语法规则 Makefile 中的第一个目标将成为默认目标，即 all 目标，但 MAKECMDGOALS 却仍是空而不是“all”，这一点值得注意。

另外，从图 3.38 所示的测试结果中还可以看出，运行 make 时可以同时指定多个目标。make 在获得了多个目标后，将以从左到右的顺序逐个地构建目标。

3．变量的类别与赋值

变量的类别有递归扩展变量和简单扩展变量。图 3.31 示例说明了使用等号进行变量定义和

赋值，这种只用一个“=”符号定义的变量被称为递归扩展变量（recursively expanded variable）。通过图 3.39 所示的 Makefile 和图 3.40 所示的运行结果，可以观察到递归扩展变量的特点。

```
.PHONY: all

foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:
    @echo $(foo)
```

图 3.39

```
$ make
Huh?
```

图 3.40

从结果来看，递归扩展变量的引用是递归的。对于图 3.41 所示的 Makefile，CFLAGS 变量最后将会被展开为“-Ifoo -Ibar -O”，而图 3.42 所示的对 CFLAGS 变量进行赋值的语句将会造成一个死循环。

```
CFLAGS = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

图 3.41

```
CFLAGS = $(CFLAGS) -O
```

图 3.42

简单扩展变量（simply expanded variable）是用“:=”操作符来定义的。对于这种变量，make 只对其进行一次展开。图 3.43 中的 Makefile 通过类比帮助我们理解其特性，运行结果如图 3.44 所示。

```
.PHONY: all

x = foo
y = $(x) b
x = later

xx := foo
yy := $(xx) b
xx := later

all :
    @echo "x = $(y), xx = $(yy)"
```

图 3.43

```
$ make
x = later b, xx = foo b
```

图 3.44

图 3.45 中的 Makefile 示例说明了对于同一个变量采用不同赋值操作的效果，其结果如图 3.46 所示。

```
.PHONY: all

objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o

all:
    @echo $(objects)
```

图 3.45

```
$ make
main.o foo.o bar.o utils.o another.o
```

图 3.46

顺便提及，在 Makefile 中还可以实现条件赋值：当变量没有被定义时就定义它，并且将右边的值赋值给它；如果变量已经定义了，则不改变其原值。条件赋值可用于为变量赋默认值。条件赋值运用条件赋值符“?=”来实现，图 3.47 和图 3.48 分别是运用条件赋值的 Makefile 和其运行结果。

```
.PHONY: all

foo = x
foo ?= y

bar ?= y
all:
    @echo "foo = $(foo), bar = $(bar)"
```

图 3.47

```
$ make
foo = x, bar = y
```

图 3.48

另一个非常有用的赋值方法是通过“+=”实现追加赋值。图 3.49 示例说明了如何使用它，其效用与图 3.45 的是完全一样的。

```
.PHONY: all

objects = main.o foo.o bar.o utils.o
objects += another.o

all:
    @echo $(objects)
```

图 3.49

4. 变量及其值的来源

从前面的示例可以看出，在 Makefile 中可以对变量进行定义。此外，还有其他的方式让 make 获得变量。比如：

（1）对于自动变量，其值是在每一个规则中根据规则的上下文自动获得的。

（2）在运行 make 时，通过命令参数定义变量。对于图 3.47 所示的 Makefile，如果以“make bar=x”的形式运行它，得到的结果则完全不同，如图 3.50 所示。从结果可以看出，在运行 make 的命令参数中定义的变量在 Makefile 中是可见的。其实，完全可以通过在 make 命令行中定义变量的方式覆盖 Makefile 文件中所定义变量的值。图 3.51 示例说明了基于图 3.47 中的 Makefile 的测试结果。

（3）变量还可以来自于 Shell 环境，图 3.52 示例说明了采用 Shell 中的 export 命令定义了一个 bar 变量后，图 3.47 中 Makefile 的运行结果。

```
$ make bar=x
foo = x, bar = x
```

图 3.50

```
$ make foo=haha
foo = haha, bar = x
```

图 3.51

```
$ make
foo = x, bar = y

$ export bar=x

$ make
foo = x, bar = x
```

图 3.52

5. 高级变量引用功能

图 3.53 中的 Makefile 示例说明了变量引用的一种高级功能，即在赋值的同时完成文件名后缀替换操作。

```
.PHONY: all

foo = a.c b.c c.c
bar := $(foo:.c=.o)

all:
	@echo "bar = $(bar)"
```

图 3.53

从图 3.54 所示的运行结果来看，bar 变量中的文件名从.c 后缀都变成了.o。与使用函数相比这种方式更加简洁。这种功能也可以采用后面将要介绍的 patsubst 函数（参见 3.2.2.4 节中的"8.patsubst 函数"小节）来实现。当然，patsubst 函数的功能更强，它还能做除后缀替换之外的其他事。

```
$ make
bar = a.o b.o c.o
```

图 3.54

6. 避免变量被覆盖的方法

图 3.51 示例说明了采用在 make 命令行上定义变量的方式，使得 Makefile 文件中定义的变量其值被覆盖。我们在设计 Makefile 时，可能并不希望发生这种覆盖现象，此时需要使用 override 指令进行预防。图 3.55 和图 3.56 分别是使用了 override 指令的 Makefile 和其运行结果。

```
Makefile
.PHONY: all

override foo = x

all:
	@echo "foo = $(foo)"
```

图 3.55

```
$ make foo=haha
foo = x
```

图 3.56

3.2.2.3　借助"模式"精简规则

对于目前 simple 项目的 Makefile，其中存在多个规则用于构建目标文件。比如，main.o 和 foo.o，都是采用不同的规则进行描述的。如果对于每一个目标文件，都得写一个不同的规则来描述，那真是一种"体力活"。Makefile 中的模式就是用来解决这种烦恼的。先看图 3.57 所示的运用了模式的 Makefile。

```
.PHONY: clean

CC = gcc
RM = rm

EXE = simple.exe
OBJS = main.o foo.o

$(EXE) : $(OBJS)
	$(CC) -o $@ $^
main.o : main.c
	$(CC) -o $@ -c $^
foo.o: foo.c
	$(CC) -o $@ -c $^
%.o : %.c
	$(CC) -o $@ -c $^
clean:
	$(RM) -fr $(EXE) $(OBJS)
```

图 3.57

与 simple 项目前一版本的 Makefile 相比，最为直观的改变就是将两条构建目标文件的规则变成了一条。模式类似于在 Windows 操作系统中所使用的通配符，用“%”加以表示。采用了模式以后，不论有多少个源文件要编译都可以应用同一条规则，这极大地简化了 Makefile。

3.2.2.4　通过“函数”增强功能

函数是 Makefile 中的另一个利器，通过使用函数能显著地增强 Makefile 的功能。对于 simple 项目的 Makefile，尽管使用了模式规则，但还有一件比较麻烦的事——在 Makefile 中要指明每一个项目源文件。

图 3.58 是采用了 wildcard（参见 3.2.2.4 节中的“11.wildcard 函数”小节）和 patsubst（参见 3.2.2.4 节中的“8. patsubst 函数”小节）两个函数后的 Makefile。读者可以先编译一下以验证其功能性。

```
.PHONY: clean

CC = gcc
RM = rm

EXE = simple.exe
SRCS = $(wildcard *.c)
OBJS = $(patsubst %.c, %.o, $(SRCS))

$(EXE) : $(OBJS)
	$(CC) -o $@ $^
%.o: %.c
	$(CC) -o $@ -c $^
clean:
	$(RM) -fr $(EXE) $(OBJS)
```

图 3.58

现在，让我们来模拟增加一个源文件的情形，看看在这种情形下不修改 Makefile 编译工作是否仍能正常完成。增加文件的方式仍然是采用 touch 命令，生成一个内容是空的 bar.c 源文件，然后运行 make 和“make clean”，结果如图 3.59 所示。

```
                /Makefile/simple
$ touch bar.c
                /Makefile/simple
$ make
gcc -o bar.o -c bar.c
gcc -o foo.o -c foo.c
gcc -o main.o -c main.c
gcc -o simple.exe bar.o foo.o main.o
                /Makefile/simple
$ make clean
rm -fr simple.exe bar.o foo.o main.o
```

图 3.59

从结果来看，增加源文件并不需要对 Makefile 进行任何的编辑。其实，不光是增加源文件，

删除源文件也同样不需要修改 Makefile。

下面的几个小节将介绍本书中需要使用到的几个函数。更多函数的使用方法请参照 make 的官方手册《GNU Make》。该手册可以从附书光盘中找到，其文件名为 make.pdf。

1. abspath 函数

abspath 函数被用于将_names 中的各路径名转换成绝对路径，并将转换后的结果返回。其形式是：

```
$(abspath _names)
```

图 3.60 示例说明了它的用法，图 3.61 是与之对应的结果。

```
.PHONY: all

ROOT := $(abspath /usr/../lib)

all:
	@echo $(ROOT)
```

图 3.60

```
$ make
/lib
```

图 3.61

2. addprefix 函数

addprefix 函数被用于给名字列表_names 中的每一个名字增加前缀_prefix，并将增加了前缀的名字列表返回。其形式是：

```
$(addprefix _prefix, _names)
```

图 3.62 示例说明了它的用法。addprefix 函数的行为可以从图 3.63 中观察到。

```
.PHONY: all

without_dir = foo.c bar.c main.o
with_dir := $(addprefix objs/, $(without_dir))

all:
	@echo $(with_dir)
```

图 3.62

```
$ make
objs/foo.c objs/bar.c objs/main.o
```

图 3.63

3. addsuffix 函数

addsuffix 函数被用于给名字列表_names 中的每一个名字增加后缀_suffix，并将增加了后缀_suffix 的名字列表返回。其形式是：

```
$(addsuffix _suffix, _names)
```

图 3.64 和图 3.65 分别示例说明了它的用法和使用效果。

```
.PHONY: all

without_suffix = foo bar main
with_suffix := $(addsuffix .c, $(without_suffix))

all:
	@echo $(with_suffix)
```

图 3.64

```
$ make
foo.c bar.c main.c
```

图 3.65

4. eval 函数

eval 函数的存在使得 Makefile 具有动态语言的特征。eval 函数使得 make 将再一次解析_text 语句。eval 函数的返回值为空字符串，其形式是：

```
$(eval _text)
```

图 3.66 示例说明了它的用法，虽然与图 3.68 的功效是完全一样的，但在某些场合却非得用 eval 函数不可。使用 eval 函数的效果，如图 3.67 所示。

```
.PHONY: all

sources = foo.c bar.c baz.s ugh.h
$(eval sources := $(filter %.c %.s, $(sources)))

all:
	@echo $(sources)
```

图 3.66

```
$ make
main.c
```

图 3.67

5. filter 函数

filter 函数被用于从一个名字列表_text 中根据模式_pattern 得到满足需要的名字列表并返回。其形式是：

```
$(filter _pattern, _text)
```

图 3.68 中的 Makefile 示例说明了它的用法，图 3.69 则是其运行结果。

```
Makefile
.PHONY: all

sources = foo.c bar.c baz.s ugh.h
sources := $(filter %.c %.s, $(sources))

all:
    @echo $(sources)
```

图 3.68

```
$ make
foo.c bar.c baz.s
```

图 3.69

从结果来看，调用 filter 函数后 source 变量中只存在.c 文件和.s 文件了，而.h 文件因为不满足所指定的模式而被过滤掉了。

6. filter-out 函数

filter-out 函数被用于从名字列表_text 中根据模式_pattern 滤除一部分名字，并将滤除后的列表返回。其形式是：

```
$(filter-out _pattern, _text)
```

图 3.70 示例说明了它的用法，图 3.71 则是其运行结果。

```
.PHONY: all

objects = main1.o foo.o main2.o bar.o
result = $(filter-out main%.o, $(objects))

all:
    @echo $(result)
```

图 3.70

```
$ make
foo.o bar.o
```

图 3.71

7. notdir 函数

notdir 函数被用来从路径_names 中抽取文件名，并将文件名返回。其形式是：

```
$(notdir _names)
```

图 3.72 示例说明了它的用法，图 3.73 则是对应的运行结果。

```
Makefile
.PHONY: all
file_name := $(notdir code/foo/src/foo.c code/bar/src/bar.c)
all:
	@echo $(file_name)
```

图 3.72

```
$ make
foo.c bar.c
```

图 3.73

8. patsubst 函数

patsubst 函数被用来将名字列表_text 中符合_pattern 模式的名字替换为_replacement，并将替换后的名字列表返回。其形式是：

```
$(patsubst _pattern, _replacement, _text)
```

图 3.74 示例说明了它的用法。从这个 Makefile 中可以看出，mixed 变量中包括了.c 文件和.o 文件，采用 patsubst 函数进行字符串替换时，希望将所有以.c 结尾的名字都替换成以.o 结尾。

```
Makefile
.PHONY: all
mixed = foo.c bar.c main.o
objects := $(patsubst %.c, %.o, $(mixed))
all:
	@echo $(objects)
```

图 3.74

图 3.75 是最后的运行结果。由于 patsubst 函数可以使用模式，所以也可以被运用于替换前缀等，其功能更强。

```
$ make
foo.o bar.o main.o
```

图 3.75

9. realpath 函数

realpath 函数被用于获取_names 所对应的真实路径名。其形式是：

```
$(realpath _names)
```

图 3.76 示例说明了它的用法，realpath 函数的运行效果如图 3.77 所示。

```
.PHONY: all

ROOT := $(realpath ./..)

all:
	@echo $(ROOT)
```

图 3.76

```
$ pwd
/home/yunli

$ make
/home
```

图 3.77

10．strip 函数

如果希望清除名字列表中的多余空格，strip 函数就是最终选择。strip 函数将_string 中的多余空格去除后返回。其形式是：

```
$(strip _string)
```

图 3.78 示例说明了它的用法，图 3.79 则是其运行结果。

```
.PHONY: all

original = foo.c    bar.c
stripped := $(strip $(original))

all:
	@echo "original = $(original)"
	@echo "stripped = $(stripped)"
```

图 3.78

```
$ make
original = foo.c    bar.c
stripped = foo.c bar.c
```

图 3.79

11．wildcard 函数

wildcard 是通配符函数，通过它可以得到当前工作目录中满足_pattern 模式的文件或目录名列表。其形式是：

```
$(wildcard _pattern)
```

图 3.80 示例说明了如何从当前 Makefile 所在的目录中通过 wildcard 函数得到所有 C 源文件的名字列表。图 3.81 则显示了最终结果。

```
.PHONY: all

SRCS = $(wildcard *.c)

all:
	@echo $(SRCS)
```

图 3.80

```
$ ls
bar.c foo.c main.c

$ make
bar.c foo.c main.c
```

图 3.81

3.3 提高编译环境的实用性

simple 项目构建了一个具有基本编译功能的编译环境，但是，正如项目的名字所暗示的那样，它只能服务于一个简单的项目。接下来，我们在 simple 项目的基础上，做一个更复杂的虚拟项目——complicated 项目。complicated 项目的初始源代码如图 3.82 所示。

```
#ifndef __FOO_H
#define __FOO_H

void foo ();

#endif
```

```
#include <stdio.h>
#include "foo.h"

void foo ()
{
   printf ("This is foo ()!\n");
}
```

```
#include "foo.h"

int main ()
{
   foo ();
   return 0;
}
```

图 3.82

3.3.1 让编译环境更加有序

大多的软件项目都会通过合理地设计目录结构来提高它的可维护性。在编译一个项目时会产生大量的中间文件，如果中间文件与项目的源程序文件直接混放在一起，就显得乱糟糟而不利于维护。

这一节我们将探讨通过使用目录让编译环境更加有序。本节中，目录的引入并不是一步到位的，我们还将在后面做 huge 项目时进一步探讨这一话题。在为 complicated 项目编写 Makefile 之前，需要先了解对目录结构的需求。包括：

（1）将所有的目标文件放入 objs 子目录中。

（2）将最终生成的可执行程序放入 exes 子目录中。

3.3.1.1 目录的自动创建与删除

在编译项目之前，需要将存放生成文件的目录准备好。目录可以在项目编译之前通过手工去创建，但我们更喜欢在编译过程中自动生成的方式。

要实现在编译过程中自动创建目录，需记住一点：目录也是一个目标。具有自动创建目录的 Makefile 和其运行结果分别如图 3.83 和图 3.84 所示。图 3.85 示例说明了 Makefile 中的规则与“依赖树”之间的映射关系。

```
.PHONY: all

MKDIR = mkdir
DIRS = objs exes

all: $(DIRS)

$(DIRS):
	$(MKDIR) $@
```

图 3.83

```
$ make
mkdir objs
mkdir exes

$ ls
Makefile  exes  objs

$ make
make: Nothing to be done for `all'.
```

图 3.84

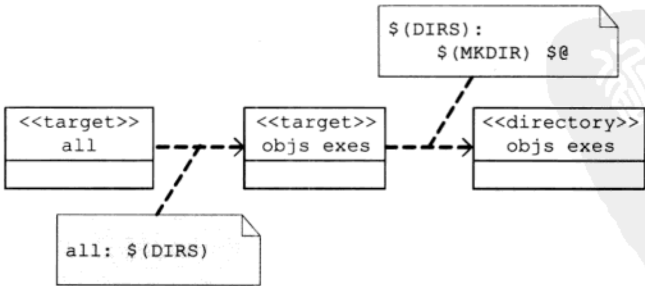


图 3.85

接下来增加一个 clean 目标用于删除 objs 及 exes 两个目录，如图 3.86 所示。这次又增加了 RM 和 RMFLAGS 两个变量。运行“make clean”命令的结果如图 3.87 所示。

```
.PHONY: all clean

MKDIR = mkdir
RM = rm
RMFLAGS = -fr

DIRS = objs exes

all: $(DIRS)

$(DIRS) :
	$(MKDIR) $@
clean :
	$(RM) $(RMFLAGS) $(DIRS)
```

图 3.86

```
$ make clean
rm -fr objs exes
```

图 3.87

3.3.1.2 通过目录管理文件

为了将项目编译时所创建的文件分别放入 objs 和 exes 目录中，需要用到 Makefile 中的一个函数——addprefix（参见 3.2.2.4 节中的“2.addprefix 函数”小节）。图 3.88 是新的 Makefile，其中包含了来自 simple 项目的内容。

```
.PHONY: all clean

MKDIR = mkdir
RM = rm
RMFLAGS = -fr

CC = gcc

DIR_OBJS = objs
DIR_EXES = exes
DIRS = $(DIR_OBJS) $(DIR_EXES)
EXE = complicated.exe
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))

all: $(DIRS) $(EXE)

$(DIRS):
	$(MKDIR) $@
$(EXE): $(OBJS)
	$(CC) -o $@ $^
$(DIR_OBJS)/%.o: %.c
	$(CC) -o $@ -c $^
clean:
	$(RM) $(RMFLAGS) $(DIRS) $(EXE)
```

图 3.88

图 3.88 中主要有三个变化：

（1）通过运用 addprefix 函数，为每一个生成的目标文件加上“objs/”前缀，以使生成的文件被放入 objs 目录中。

（2）在构建目标文件的规则中为目标名加上“objs/”前缀，即增加“$（DIR_OBJS）/”前缀。

（3）在 clean 规则的命令中增加对$（EXE）目标的删除。

更改以后 Makefile 的运行结果如图 3.89 所示。从项目编译过程可以看到，所生成的目标文件已放入了 objs 子目录中。

```
/Makefile/complicated
$ make
mkdir objs
mkdir exes
cc -o objs/foo.o -c foo.c
cc -o objs/main.o -c main.c
cc -o complicated.exe objs/foo.o objs/main.o
/Makefile/complicated
$ ls
Makefile  complicated.exe  exes/  foo.c  foo.h  main.c  objs/
```

图 3.89

采用同样的方法，可以将 complicated.exe 放入到 exes 目录中，这里不再详述。

3.3.2　提升依赖关系管理

现在假设对项目已经进行了一次成功的编译，这一点非常重要，否则看不到现有 Makefile 所存在的问题。接着，将 foo.h 文件更改为如图 3.90 所示，但不修改 foo.c 文件。理论上，由于 foo()函数的声明与定义不相同，编译时应出错。

```
#ifndef __FOO_H
#define __FOO_H

void foo (int _value);

#endif
```

图 3.90

图 3.91 示例说明了更改 foo.h 后的 make 结果。对 make 告诉我们没有什么事可做是不是很吃惊？

```
$ make
make: Nothing to be done for `all'.
```

图 3.91

此时进行“make clean”，然后再 make 又是什么结果呢？答案从图 3.92 中可以找到。

```
                     /Makefile/complicated
$ make clean
rm -fr objs exes
                     /Makefile/complicated
$ make
mkdir objs
mkdir exes
gcc -o objs/foo.o -c foo.c
gcc -o objs/main.o -c main.c
main.c: In function 'main':
main.c:5: error: too few arguments to function 'foo'
make: *** [objs/main.o] Error 1
```

图 3.92

为什么在不进行“make clean”之前，make 却没有发现需要对项目进行重新构建呢？

在改进之前，让我们分析一下 make 为什么不能发现 foo.h 的更改并对项目进行重新编译。图 3.93 所示是现有 Makefile 所表达的依赖关系树。从图中我们并不能找到 foo.h 文件的身影，也就是说，从 make 的角度来看它并不知道 foo.h 的存在，因而也不可能侦测到 foo.h 文件的变动并对项目进行重新编译。

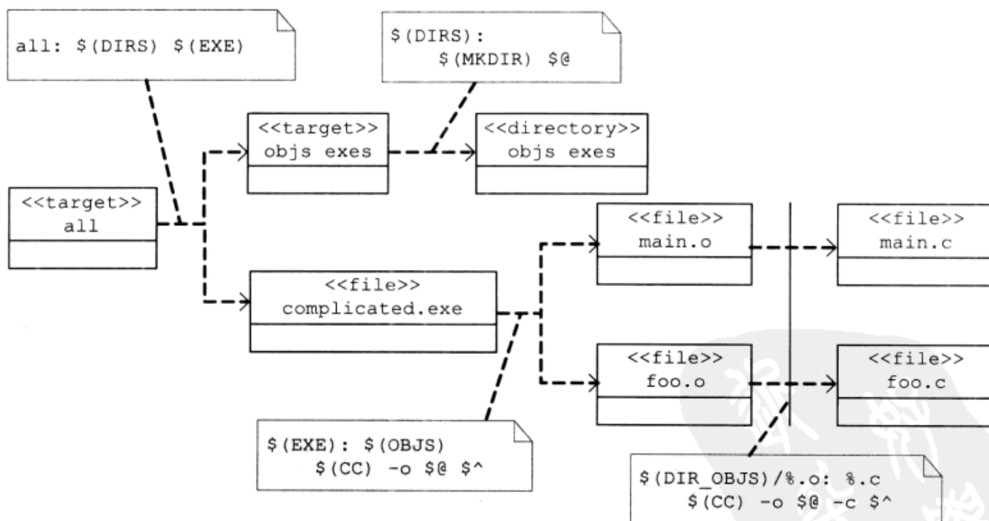


图 3.93

为了改进 Makefile，最为直接的方法就是将 foo.h 文件通过依赖关系树纳入 make 的视野中。改动后的 Makefile 如图 3.94 所示。

```
.PHONY: all clean

MKDIR = mkdir
```

```
RM = rm
RMFLAGS = -fr

CC = gcc

DIR_OBJS = objs
DIR_EXES = exes
DIRS = $(DIR_OBJS) $(DIR_EXES)
EXE = complicated.exe
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))

all: $(DIRS) $(EXE)

$(DIRS):
	$(MKDIR) $@
$(EXE): $(OBJS)
	$(CC) -o $@ $^
$(DIR_OBJS)/%.o: %.c foo.h
	$(CC) -o $@ -c $<
clean:
	$(RM) $(RMFLAGS) $(DIRS)
```

图 3.94

其中的改动非常小，即将 foo.h 文件作为每一个.o 文件的先决条件。在这个 Makefile 中，首次使用了自动变量$<。用$<的目的是为了只将.c 文件作为 gcc 的输入内容。

有了这样的 Makefile 后，对现有 complicated 项目中的任何文件进行更改，make 都能正确地识别出改动并重新构建相关目标。

虽然现在的 Makefile 能正常工作，但解决问题的方法却不具可操作性。当项目复杂时，如果要将每一个头文件都写入到 Makefile 的相应规则中，会是另一个恶梦。看来，我们还得寻找另一种更好的解决方法。

3.3.2.1 自动生成文件依赖关系

在 4.3.6 节介绍了如何通过 gcc 获得一个源文件对其他依赖文件的列表，gcc 的这个功能其实就是为 make 而存在的。下面看看如何将之运用于 complicated 项目的 Makefile 中。

图 3.95 是采用 gcc 的-MM 选项并结合 sed 命令后的输出结果。使用 sed 进行替换的目的是为了在目标名前加上“objs/”前缀。

```
$ gcc -MM foo.c | sed 's,\(.*\)\.o[ :]*,objs/\1.o: ,g'
objs/foo.o: foo.c foo.h
```

图 3.95

gcc 还有另一个非常有用的-E 选项（参见 4.3.1 节），这个选项告诉 gcc 只做预处理而不进行程序编译。在生成依赖关系时，其实并不需要 gcc 编译源文件，只要进行预处理以获得所依

赖文件的列表就行了。通过使用-E 选项，可以避免生成依赖关系时 gcc 发出编译警告，以及提高依赖关系的生成效率。

现在，已经找到了自动生成依赖关系的方法了，那如何将其整合到 complicated 项目的 Makefile 中呢？自动生成的依赖信息不可能直接出现在 Makefile 中，因为不能动态地改变 Makefile 中的内容。此时我们需要通过创建依赖关系文件的方式。

第一步能做的事是为每一个源文件通过采用 gcc 和 sed 生成一个依赖关系文件，这些文件假设采用".dep"后缀结尾。在此，创建一个新的 deps 目录用于存放生成的依赖关系文件。图 3.96 中的 Makefile 增加了创建 deps 目录和为每一个源文件生成依赖关系文件的规则。

```
.PHONY: all clean

MKDIR = mkdir
RM = rm
RMFLAGS = -fr

CC = gcc

DIR_OBJS = objs
DIR_EXES = exes
DIR_DEPS = deps
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS)
EXE = complicated.exe
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

all: $(DIRS) $(DEPS) $(EXE)

$(DIRS):
	$(MKDIR) $@
$(EXE): $(OBJS)
	$(CC) -o $@ $^
$(DIR_OBJS)/%.o: %.c foo.h
	$(CC) -o $@ -c $^
$(DIR_DEPS)/%.dep: %.c
	@echo "Creating $@ ..."
	@set -e; \
	$(RM) $(RMFLAGS) $@.tmp ; \
	$(CC) -E -MM $^ > $@.tmp ; \
	sed 's,\(.*\)\.o[ :]*,objs/\1.o: ,g' < $@.tmp > $@ ; \
	$(RM) $(RMFLAGS) $@.tmp
clean:
	$(RM) $(RMFLAGS) $(DIRS)
```

图 3.96

图 3.96 中的 Makefile 包含以下更改。

（1）增加了 DIR_DEPS 变量用于保存需要创建的 deps 目录名，以及将这个变量的值加入

到 DIR 变量中。

（2）删除了目标文件创建规则中对于 foo.h 文件的依赖，并将这个规则中的自动变量从$<变回了$^。

（3）增加了 DEPS 变量用于存放依赖文件。

（4）为 all 目标增加了对$（DEPS）的依赖。

（5）增加了一个用于创建依赖关系文件的规则。在这个规则的命令中，使用了 gcc 的-E 和-MM 选项来获取依赖关系。在生成最终的依赖关系文件之前，使用了一个由$@.tmp 表示的临时文件，且在依赖关系文件生成以后将其删除。在这个规则中，“set -e”的作用是告诉 Shell，在生成依赖关系文件的过程中如果出现任何错误就直接退出。Shell 异常退出的最终表现就是 make 会告诉我们出错了，从而停止后续的 make 工作。如果不进行这一设置，当构建依赖文件出现错误时，make 还会继续后面的工作（并最终出错），这并不是我们所希望的。读者可以试着将“set -e”去掉，且故意在 foo.c 或者 main.c 中植入错误，以观察 make 生成依赖关系时的行为有何不同。

这里又有几个知识点需要掌握。

（1）对于规则中的每一条命令，make 都是在一个新的 Shell 上运行它的。

（2）如果希望多个命令在同一个 Shell 中运行，可以用“;”将这些命令连起来。

（3）当命令很长时，可以用“\”将一个命令分成多行书写。

为了帮助理解第 1 个知识点，我们可以做一个实验。现在假设需要创建一个 test 目录，然后在这个 test 目录下再创建一个 subtest 子目录。如果写一个如图 3.97 所示那样的 Makefile，并不能达到目的。

```
.PHONY: all

all:
	@mkdir test
	@cd test
	@mkdir subtest
```

图 3.97

图 3.98 是该 Makefile 的运行结果。从 ls 结果来看，make 在当前目录中创建了 test 和 subtest 两个目录，即 test 与 subtest 目录是同级的，而非父子关系。

```
$ make

$ ls
Makefile  subtest/  test/
```

图 3.98

现在，通过运用前面提到的知识点，将 Makefile 修改为如图 3.99 所示那样。

```
.PHONY: all

all:
	@mkdir test ; \
	cd test ; \
	mkdir subtest
```

图 3.99

在手动删除前面创建的 test 和 subtest 目录后，再次运行 make 并查看结果。最后的结果可以从图 3.100 中找到，这次的目录结构与所希望的完全相同。

```
$ rm -fr subtest/ test/

$ make

$ ls
Makefile  test/

$ ls test
subtest/
```

图 3.100

回到 complicated 项目的 Makefile 上，图 3.101 是图 3.96 中 Makefile 的运行结果。从图中可以看到，Makefile 会生成新的目录 deps、创建 foo.dep 和 main.dep 依赖文件。从使用 cat 命令查看的结果来看，两个依赖文件的内容正是我们所希望的。

```
                       /Makefile/complicated
$ make
mkdir objs
mkdir exes
mkdir deps
Creating deps/foo.dep ...
Creating deps/main.dep ...
gcc -o objs/foo.o -c foo.c
gcc -o objs/main.o -c main.c
gcc -o exes/complicated.exe objs/foo.o objs/main.o
                       /Makefile/complicated
$ cat deps/foo.dep
objs/foo.o: foo.c foo.h
                       /Makefile/complicated
$ cat deps/main.dep
objs/main.o: main.c foo.h
```

图 3.101

3.3.2.2　使用依赖关系文件

Makefile 中存在一个 include 指令，它的作用如同 C 语言中的#include 宏指令。在 Makefile 中，可以通过使用 include 指令将自动生成的依赖关系文件包含进来，从而使得依赖关系文件

中的内容成为 Makefile 的一部分。

使用 include 指令包含自动生成的依赖关系文件的 Makefile 如图 3.102 所示。

```
complicated/Makefile
.PHONY: all clean

MKDIR = mkdir
RM = rm
RMFLAGS = -fr

CC = gcc

DIR_OBJS = objs
DIR_EXES = exes
DIR_DEPS = deps
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS)

EXE = complicated.exe
EXE := $(addprefix $(DIR_EXES)/, $(EXE))

SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

all: $(DIRS) $(DEPS) $(EXE)

include $(DEPS)

$(DIRS):
	$(MKDIR) $@
$(EXE): $(OBJS)
	$(CC) -o $@ $^
$(DIR_OBJS)/%.o: %.c
	$(CC) -o $@ -c $<
$(DIR_DEPS)/%.dep: %.c
	@echo "Creating $@ ..."
	@set -e; \
	$(RM) $(RMFLAGS) $@.tmp ; \
	$(CC) -E -MM $^ > $@.tmp ; \
	sed 's,\(.*\)\.o[ :]*,objs/\1.o: ,g' < $@.tmp > $@ ; \
	$(RM) $(RMFLAGS) $@.tmp
clean:
	$(RM) $(RMFLAGS) $(DIRS)
```

图 3.102

运行 make 所获得的运行结果如图 3.103 所示。

```
/Makefile/complicated
$ make
Makefile:25: deps/foo.dep: No such file or directory
Makefile:25: deps/main.dep: No such file or directory
Creating deps/main.dep ...
/bin/sh: line 2: deps/main.dep.tmp: No such file or directory
make: *** [deps/main.dep] Error 1
```

图 3.103

从运行结果来看出错了。错误发生的原因是：

（1）make 在处理 Makefile 中的 include 命令时，发现找不到 deps/foo.dep 和 deps/main.dep 文件。

（2）由于 Makefile 中包含了创建依赖关系文件的规则，所以 make 试图使用规则去创建依赖关系文件。

（3）由于 deps 目录是在构建 all 目标时才创建的，所以 make 在处理 include 指令而创建依赖文件时，由于 deps 目录不存在，因此出现了不能创建依赖关系文件的错误。

明白了出错的原因后，就可以对依赖关系进行调整，将 deps 目录的创建放在构建依赖文件之前。改动就是在依赖文件的创建规则中增加对 deps 目录的依赖，且将其当做第一个先决条件。采用同样的方法，将所有的目录创建都放到相应的规则中。更改后的 Makefile 如图 3.104 所示。

```
.PHONY: all clean

MKDIR = mkdir
RM = rm
RMFLAGS = -fr

CC = gcc

DIR_OBJS = objs
DIR_EXES = exes
DIR_DEPS = deps
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS)

EXE = complicated.exe
EXE := $(addprefix $(DIR_EXES)/, $(EXE))

SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

all: $(DIRS) $(DEPS) $(EXE)

include $(DEPS)

$(DIRS):
	$(MKDIR) $@
$(EXE): $(DIR_EXES) $(OBJS)
	$(CC) -o $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DIR_OBJS) %.c
	$(CC) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DIR_DEPS) %.c
	@echo "Creating $@ ..."
	@set -e; \
	$(RM) $(RMFLAGS) $@.tmp ; \
	$(CC) -E -MM $(filter %.c, $^) > $@.tmp ; \
```

```
    sed 's,\(.*\)\.o[ :]*,objs/\1.o: ,g' < $@.tmp > $@ ; \
    $(RM) $(RMFLAGS) $@.tmp
clean:
    $(RM) $(RMFLAGS) $(DIRS)
```

图 3.104

这次的改动用到了 filter 函数，将规则所依赖的目录从先决条件中去除。读者可以试试看，如果不使用 filter 函数会出现什么错误。

正如前面所提及的，当 make 看到 include 指令时会试图去构建所需包含进来的依赖文件，如此一来，我们就不需要显式地让 all 目标依赖于它们了。这也是为什么这次修改从 all 规则中去除了对$（DEPS）的依赖。图 3.105 示例说明了修订后的 Makefile 的运行结果。

```
                    /Makefile/complicated
$ make
mkdir deps
Creating deps/main.dep ...
Creating deps/foo.dep ...
mkdir exes
mkdir objs
gcc -o objs/foo.o -c foo.c
gcc -o objs/main.o -c main.c
gcc -o exes/complicated.exe objs/foo.o objs/main.o
                    /Makefile/complicated
$ make clean
rm -fr objs exes deps
```

图 3.105

需要特别指出的是，获得这一结果的 Cygwin 是安装在 FAT32 文件系统之上的。如果读者的 Cygwin 是安装在 NTFS 文件系统之上，或者是在某些 Linux 操作系统之上的，这个 Makefile 将不能正常工作，会出现像图 3.106 那样的死循环。

```
                    /Makefile/complicated
$ make
Creating deps/main.dep ...
Creating deps/foo.dep ...
Creating deps/main.dep ...
Creating deps/foo.dep ...
Creating deps/main.dep ...
```

图 3.106

出现死循环的原因与文件系统的实现有关。有些文件系统当目录中的文件被更改时，目录的时间戳也会随之更新。在这样的文件系统上，make 生成了 main.dep 和 foo.dep 后将导致 deps 目录的时间戳也随之更改。由于 Makefile 创建依赖关系文件的规则中，指定了 deps 目录是其第一个先决条件，于是，deps 目录时间戳的更改使得 make 又一次使用规则以再一次创建 main.dep 和 foo.dep，如此循环往复就导致了死循环。

为了在这些文件系统中解决死循环问题，必须再一次修改规则的依赖关系。思路是：

（1）如果 deps 目录不存在，则让 deps 目录成为规则的第一个先决条件。

（2）如果 deps 目录已经存在，则不要让 deps 目录出现在规则的先决条件中。

要沿着这样的思路走下去，需要运用 Makefile 中的条件语法。

3.3.2.3 运用条件语法

当 make 看到条件语法时将立即对其进行分析，包括 ifdef、ifeq、ifndef 和 ifneq 四种语句形式。Makefile 中的条件语法有三种形式，如图 3.107 所示。其中的 conditional-directive 可以是 ifdef、ifeq、ifndef 和 ifneq 中的任意一个。

```
conditional-directive
text-if-true
endif
```

```
conditional-directive
   text-if-true
else
   text-if-false
endif
```

```
conditional-directive
   text-if-one-is-true
else conditional-directive
   text-if-true
else
   text-if-false
endif
```

图 3.107

有了条件语法以后，可以将 complicated 项目的 Makefile 改写成图 3.108 那样。

```
.PHONY: all clean

MKDIR = mkdir
RM = rm
RMFLAGS = -fr

CC = gcc

DIR_OBJS = objs
DIR_EXES = exes
DIR_DEPS = deps
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS)

EXE = complicated.exe
EXE := $(addprefix $(DIR_EXES)/, $(EXE))

SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))
```

```
ifeq ("$(wildcard $(DIR_OBJS))", "")
DEP_DIR_OBJS := $(DIR_OBJS)
endif
ifeq ("$(wildcard $(DIR_EXES))", "")
DEP_DIR_EXES := $(DIR_EXES)
endif
ifeq ("$(wildcard $(DIR_DEPS))", "")
DEP_DIR_DEPS := $(DIR_DEPS)
endif

all: $(EXE)

include $(DEPS)

$(DIRS):
	$(MKDIR) $@
$(EXE): $(DEP_DIR_EXES) $(OBJS)
	$(CC) -o $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DEP_DIR_OBJS) %.c
	$(CC) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DEP_DIR_DEPS) %.c
	@echo "Creating $@ ..."
	@set -e; \
	$(RM) $(RMFLAGS) $@.tmp ; \
	$(CC) -E -MM $(filter %.c, $^) > $@.tmp ; \
	sed 's,\(.*\)\.o[ :]*,objs/\1.o: ,g' < $@.tmp > $@ ; \
	$(RM) $(RMFLAGS) $@.tmp
clean:
	$(RM) $(RMFLAGS) $(DIRS)
```

图 3.108

改动主要是增加了三个变量，这三个变量的值根据相应的目录是否存在而分别赋值。对于每一个变量，如果对应的目录不存在，则将目录名赋值给它，否则其值为空。增加的三个变量，分别被放到相应的规则中作为第一个先决条件。修改后 Makefile 的检验结果如图 3.109 所示。死循环的问题得到了解决。

```
/Makefile/complicated
$ make
Makefile:35: deps/foo.dep: No such file or directory
Makefile:35: deps/main.dep: No such file or directory
mkdir deps
Creating deps/main.dep ...
Creating deps/foo.dep ...
mkdir exes
mkdir objs
gcc -o objs/foo.o -c foo.c
gcc -o objs/main.o -c main.c
gcc -o exes/complicated.exe objs/foo.o objs/main.o
/Makefile/complicated
$ make clean
rm -fr objs exes deps
```

图 3.109

3.3.2.4　为依赖关系文件建立依赖关系

现在，让我们再对 complicated 项目的源程序文件进行一定的修改，以便增加程序文件间

依赖关系的复杂度，如图 3.110 所示。

```
#ifndef __DEFINE_H
#define __DEFINE_H

#define HELLO "Hello"

#endif
```

```
#ifndef __FOO_H
#define __FOO_H
#include "define.h"

void foo ();

#endif
```

```
#include <stdio.h>
#include "foo.h"

void foo ()
{
   printf ("%s, this is foo ()!\n", HELLO);
}
```

```
#include "foo.h"

int main ()
{
   foo ();
   return 0;
}
```

图 3.110

其中的改动包括：

（1）增加 define.h 文件并在其中定义一个 HELLO 宏。

（2）在 foo.h 中包含 define.h 文件。

（3）在 foo.c 中增加对 HELLO 宏的引用。

增加了这些改动以后，对项目进行一次 make，结果如图 3.111 所示。

```
$ mkdir deps
Creating deps/main.dep ...
Creating deps/foo.dep ...
mkdir exes
mkdir objs
gcc -o objs/foo.o -c foo.c
gcc -o objs/main.o -c main.c
gcc -o exes/complicated.exe objs/foo.o objs/main.o

$ cat deps/foo.dep
objs/foo.o: foo.c foo.h define.h
```

图 3.111

在这次成功编译项目的基础上，我们再做一些改动。注意，一定不要在改动之前运行“make clean”，否则不能发现新的问题。

改动如图 3.112 所示，这次增加了 other.h 文件并将以前在 define.h 文件中定义的 HELLO 宏放到了这个文件中。另外，让 define.h 包含 other.h 文件。引入这些变更后，再进行一次 make 操作，结果如图 3.113 所示。

```
#ifndef __DEFINE_H
#define __DEFINE_H
#include "other.h"
#endif
```

```
#ifndef __OTHER_H
#define __OTHER_H

#define HELLO "Hello"

#endif
```

图 3.112

```
/Makefile/complicated
$ make
gcc -o objs/foo.o -c foo.c
gcc -o objs/main.o -c main.c
gcc -o exes/complicated.exe objs/foo.o objs/main.o
/Makefile/complicated
$ ./exes/complicated.exe
Hello, this is foo ()!
```

图 3.113

从结果来看，尽管 foo.c 和 main.c 文件被重新编译了，但依赖关系文件却没有被重新构建。从运行 complicated 程序的运行结果来看，其打印的问候语是所希望的“Hello”。

现在对 other.h 文件再进行更改，将问候语从“Hello”变成“Hi”，更改后的内容如图 3.114 所示。运行 make 的结果如图 3.115 所示。很明显，项目并没有因为更改了 other.h 文件而重新编译。

```
#ifndef __OTHER_H
#define __OTHER_H

#define HELLO "Hi"

#endif
```

图 3.114

```
/Makefile/complicated
$ make
make: Nothing to be done for `all'.
/Makefile/complicated
```

```
$ cat deps/foo.dep
objs/foo.o: foo.c foo.h define.h
                    /Makefile/complicated
$ cat deps/main.dep
objs/main.o: main.c foo.h define.h
```

图 3.115

理论上，当前面的 define.h 文件被更改为包含 other.h 文件后，foo.dep 和 main.dep 文件也应当重新被生成以反映出 foo.o 和 main.o 文件对 other.h 文件的依赖。正是由于依赖关系中没有正确地反映出对 other.h 文件的依赖，所以当对 other.h 文件进行更改时，make 不能发现需要对项目进行重新编译。

完善的方法是，为 foo.dep 和 main.dep 也引入图 3.116 所示的依赖关系，这个依赖关系图假设 other.h 文件还没有加入到项目中。有了这种依赖关系以后，前面一旦对 define.h 文件进行了更改，make 就能发现需要对依赖关系文件进行重新构建，而这将造成 other.h 文件出现在项目的依赖关系树中。

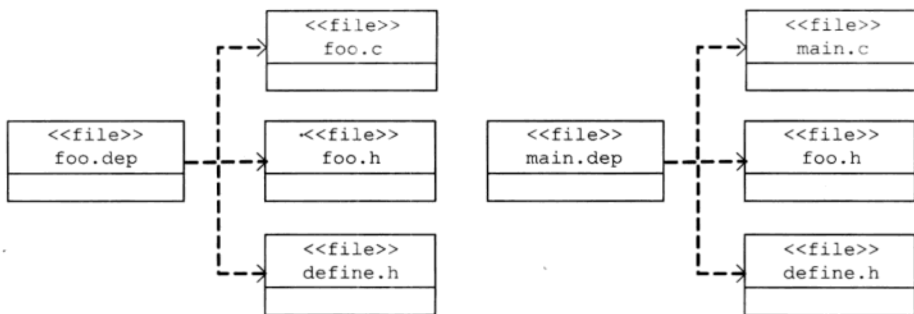


图 3.116

要在现有的 Makefile 上增加这样的依赖关系是一件很简单的事。图 3.117 示例说明了改动的原理，只要在生成的依赖关系中增加依赖文件的名称就行了。更改后的 Makefile 如图 3.118 所示。

```
                    /Makefile/complicated
$ gcc -MM foo.c | sed 's,\(.*\)\.o[ :]*,objs/\1.o deps/foo.dep: ,g'
objs/foo.o deps/foo.dep: foo.c foo.h define.h
```

图 3.117

```
.PHONY: all clean

MKDIR = mkdir
RM = rm
RMFLAGS = -fr

CC = gcc

DIR_OBJS = objs
DIR_EXES = exes
DIR_DEPS = deps
```

```
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS)

EXE = complicated.exe
EXE := $(addprefix $(DIR_EXES)/, $(EXE))

SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

ifeq ("$(wildcard $(DIR_OBJS))", "")
DEP_DIR_OBJS := $(DIR_OBJS)
endif
ifeq ("$(wildcard $(DIR_EXES))", "")
DEP_DIR_EXES := $(DIR_EXES)
endif
ifeq ("$(wildcard $(DIR_DEPS))", "")
DEP_DIR_DEPS := $(DIR_DEPS)
endif

all: $(EXE)

include $(DEPS)

$(DIRS):
	$(MKDIR) $@
$(EXE): $(DEP_DIR_EXES) $(OBJS)
	$(CC) -o $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DEP_DIR_OBJS) %.c
	$(CC) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DEP_DIR_DEPS) %.c
	@echo "Creating $@ ..."
	@set -e; \
	$(RM) $(RMFLAGS) $@.tmp ; \
	$(CC) -E -MM $(filter %.c, $^) > $@.tmp ; \
	sed 's,\(.*\)\.o[ :]*,objs/\1.o $@: ,g' < $@.tmp > $@ ; \
	$(RM) $(RMFLAGS) $@.tmp
clean:
	$(RM) $(RMFLAGS) $(DIRS)
```

图 3.118

在 Makefile 中只需在构建依赖关系文件的规则中增加自动变量$@就行了，因为它表示的是依赖关系文件名。有了这样的改动后，不能直接进行 make 来验证其效果，而是必须先运行“make clean”，然后依次重新做图 3.112 和图 3.114 中所列出的变化。

现在的 Makefile，我们更改项目中的任意一个文件，make 都能发现并做出正确的重构响应。但是，在项目已编译好的情形下，如果连续运行两次“make clean”，读者将发现两次的输出结果并不相同，如图 3.119 所示。

```
$ make clean
rm -fr objs exes deps
$ make clean
mkdir deps
```

```
Creating deps/main.dep ...
Creating deps/foo.dep ...
rm -fr objs exes deps
```

图 3.119

第一次运行“make clean”时，make 直接调用 rm 命令删除相应的目录；而第二次运行“make clean”时，make 先会构建依赖关系文件，紧接着又将所有的目录给删除了，当然包括刚生成的依赖关系文件。为什么第二次运行“make clean”时 make 会构建依赖关系文件，相信读者能解释这一现象。

为了去除在运行“make clean”时不必要的依赖关系文件构建动作，可以再一次运用条件语法。方法是：当运行“make clean”时让 Makefile 不将依赖关系文件包含进去。更改后的 Makefile 如图 3.120 所示。在这次修改中我们用到了 MAKECMDGOALS 变量。

```
.PHONY: all clean

MKDIR = mkdir
RM = rm
RMFLAGS = -fr

CC = gcc

DIR_OBJS = objs
DIR_EXES = exes
DIR_DEPS = deps
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS)

EXE = complicated.exe
EXE := $(addprefix $(DIR_EXES)/, $(EXE))

SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

ifeq ("$(wildcard $(DIR_OBJS))", "")
DEP_DIR_OBJS := $(DIR_OBJS)
endif
ifeq ("$(wildcard $(DIR_EXES))", "")
DEP_DIR_EXES := $(DIR_EXES)
endif
ifeq ("$(wildcard $(DIR_DEPS))", "")
DEP_DIR_DEPS := $(DIR_DEPS)
endif

all: $(EXE)

ifneq ($(MAKECMDGOALS), clean)
include $(DEPS)
endif

$(DIRS):
    $(MKDIR) $@
```

```
$(EXE): $(DEP_DIR_EXES) $(OBJS)
	$(CC) -o $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DEP_DIR_OBJS) %.c
	$(CC) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DEP_DIR_DEPS) %.c
	@echo "Creating $@ ..."
	@set -e; \
	$(RM) $(RMFLAGS) $@.tmp ; \
	$(CC) -E -MM $(filter %.c, $^) > $@.tmp ; \
	sed 's,\(.*\)\.o[ :]*,objs/\1.o $@: ,g' < $@.tmp > $@ ; \
	$(RM) $(RMFLAGS) $@.tmp
clean:
	$(RM) $(RMFLAGS) $(DIRS)
```

图 3.120

读者可能还存在一个困惑——在生成的依赖关系文件中，其中的规则只描述了依赖关系，而没有任何的命令。make 是如何知道使用哪些命令进行目标构建呢？这与 Makefile 的另一个特性有关。

当一个 Makefile 中存在构建同一目标的不同规则时，make 会将这些规则合在一起，合并的内容包括先决条件和命令。尽管在自动生成的依赖关系文件中只存在目标和先决条件，但由于在 Makefile 中已经定义了.o 和.dep 文件的生成规则，因此 make 会将这两部分合在一起，从而形成最终针对一个（类）构建目标的规则。图 3.121 中的 Makefile 能很好地帮助理解 make 的这一特性，其运行结果示于图 3.122 中。

```
all:
	@echo "command of rule"

all: dep

dep:
	@echo "prerequisite of rule"
```

图 3.121

```
$ make
command of rule
prerequisite of rule
```

图 3.122

3.4　打造更专业的编译环境

在第 16 章中我们会谈到，一个好的目录结构对于软件项目的维护至关重要，而 Makefile 的设计也应当迎合项目目录结构规划的需要。前面的 simple 和 complicated 项目，对于源文件我们采用了单一的目录结构，但大型项目往往用多个目录以存放不同的模块。下面我们通过虚拟的 huge 项目来实现一个更加专业的编译环境。让我们从项目的目录结构规划开始。

3.4.1 规划项目目录结构

图 3.123 示例说明了 huge 项目将采用的目录结构。从图中可以看出，huge 项目最上层有两个目录，其中一个是 build 目录，另一个是 code 目录。前者用于存放各 Makefile 文件间的共享文件 make.rule，以及编译整个项目的的 Makefile。在 build 目录中还会在编译期间自动生成 libs 和 exes 两个子目录。libs 目录用于存放编译出来的目标文件，而 exes 目录用于存放编译出来的可执行文件。

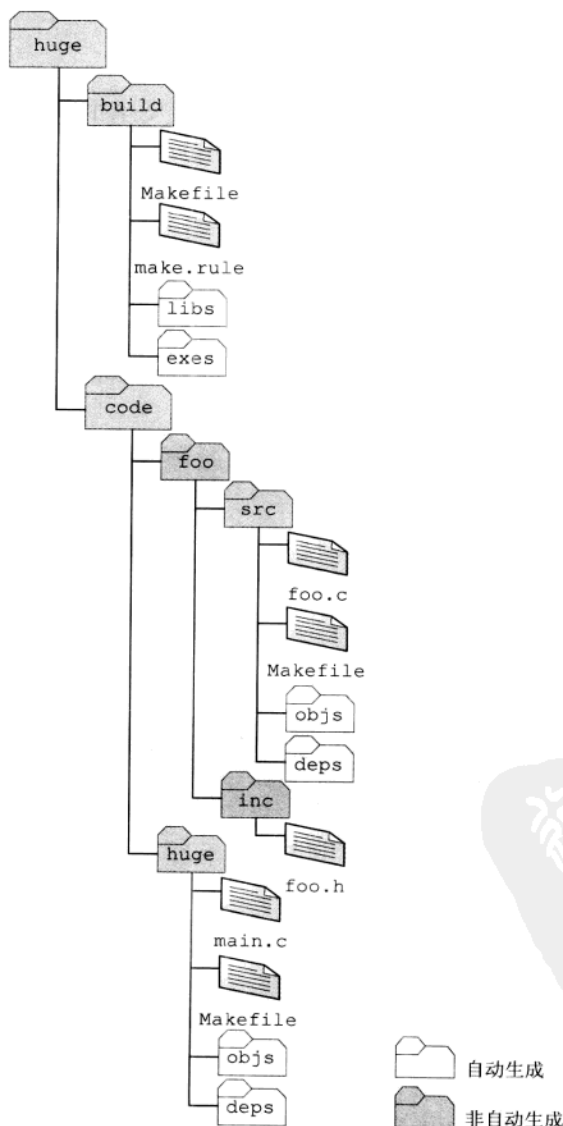


图 3.123

code 目录则用于存放项目的源程序文件，其下将按各个软件模块分成不同的子目录。huge 项目中包括 foo 库和 huge 主程序，所以在 code 目录下分别创建了 foo 和 huge 两个子目录。

对于每个软件模块子目录，又分为用于存放.c 文件的 src 子目录和用于存放.h 文件的 inc 子目录。当进行项目编译时，我们希望 make 在 src 目录下面创建 deps 和 objs 目录，两个目录的作用与 complicated 项目中的完全一样。

在每一个 src 目录中都会有一个 Makefile，用于构建所在目录中的源程序文件。可以推测，在 build 目录下面的 Makefile，将调用每一个软件模块中 src 子目录内的 Makefile，从而完成整个项目的构建。

我们需要先根据图 3.123 创建好那些需要手动完成的目录，采用图 3.124 所示的命令能完成这些目录的创建工作。

```
/Makefile/huge
$ mkdir -p build code/foo/src code/foo/inc code/huge/src
```

图 3.124

接下来，假设先创建位于 code/foo/src 目录下的 Makefile，它可以基于 complicated 项目最终版本的 Makefile 进行一定的修改而获得，如图 3.125 所示。

```
.PHONY: all clean

MKDIR = mkdir
RM = rm
RMFLAGS = -fr

CC = gcc
AR = ar
ARFLAGS = crs

DIR_OBJS = objs
DIR_EXES = ../../../build/exes
DIR_DEPS = deps
DIR_LIBS = ../../../build/libs
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS) $(DIR_LIBS)
RMS = $(DIR_OBJS) $(DIR_DEPS)

EXE = complicated.exe
ifneq ("$(EXE)", "")
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
RMS += $(EXE)
endif

LIB = libfoo.a
ifneq ("$(LIB)", "")
LIB := $(addprefix $(DIR_LIBS)/, $(LIB))
RMS += $(LIB)
endif

SRCS = $(wildcard *.c)
```

```
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

ifeq ("$(wildcard $(DIR_OBJS))", "")
DEP_DIR_OBJS := $(DIR_OBJS)
endif
ifeq ("$(wildcard $(DIR_EXES))", "")
DEP_DIR_EXES := $(DIR_EXES)
endif
ifeq ("$(wildcard $(DIR_DEPS))", "")
DEP_DIR_DEPS := $(DIR_DEPS)
endif
ifeq ("$(wildcard $(DIR_LIBS))", "")
DEP_DIR_LIBS := $(DIR_LIBS)
endif

all: $(EXE) $(LIB)

ifneq ($(MAKECMDGOALS), clean)
include $(DEPS)
endif

$(DIRS):
	$(MKDIR) $@
$(EXE): $(DEP_DIR_EXES) $(OBJS)
	$(CC) -o $@ $(filter %.o, $^)
$(LIB): $(DEP_DIR_LIBS) $(OBJS)
	$(AR) $(ARFLAGS) $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DEP_DIR_OBJS) %.c
	$(CC) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DEP_DIR_DEPS) %.c
	@echo "Creating $@ ..."
	@set -e ; \
	$(RM) $(RMFLAGS) $@.tmp ; \
	$(CC) -E -MM $(filter %.c, $^) > $@.tmp ; \
	sed 's,\(.*\)\.o[ :]*,objs/\1.o $@: ,g' < $@.tmp > $@ ; \
	$(RM) $(RMFLAGS) $@.tmp
clean:
	$(RM) $(RMFLAGS) $(DIRS) $(RMS)
```

图 3.125

其中包含如下更改。

（1）增加了 AR 和 ARFLAGS 两个变量，它们被用于静态库的创建。ar 工具的使用请参见 5.2 节。

（2）将 exes 目录的实际位置以相对路径的形式赋值给 DIR_EXES 变量。

（3）增加了 DIR_LIBS 变量以记录 libs 目录的实际位置，同样采用相对路径的形式。

（4）在 DIRS 变量中增加了 DIR_LIBS 变量的值，以便创建 build/libs 目录。

（5）新增了 RMS 变量用于表示需要删除的目录和（或）文件。由于这个 Makefile 只是针

对构建 libfoo.a 库的，所以当运行“make clean”时，不应将位于 build 目录下的 exes 和 libs 目录全部删除，这与 complicated 项目很不一样。

（6）清除了对 EXE 变量所赋的 complicated.exe 值，同时增加了 ifneq 条件语句用于判断 EXE 变量的值是否为空。只有当 EXE 不为空时才需要为 EXE 变量的值增加目录前缀并将$（EXE）加入到 RMS 变量中，以便在调用“make clean”时清除它。

（7）新增了 LIB 变量，用于存放最终生成库的名字，目前这个值被设置为 libfoo.a。同样采用处理 EXE 变量的方法，使用条件语法来决定是否需要为 LIB 变量中的值增加目录前缀。

（8）为 all 目标增加$（LIB）先决条件。

（9）增加了一条用于生成库的规则，在规则的命令体中使用 ar 工具来生成库。

（10）在 clean 目标命令中，采用删除 RMS 变量中的内容而不是 DIRS 变量中的内容的方式。这一点前面说过了，因为我们不希望在 foo 模块中运行“make clean”时将 build 目录下的 libs 和 exes 目录也删除。

现在试一试这个 Makefile 是否能工作，在试之前，需要先在 src 目录中增加一个源文件。这可以采用 touch 命令来创建一个空的 foo.c 文件，操作结果如图 3.126 所示。

```
                  /Makefile/huge/code/foo/src
$ touch foo.c

$ make
mkdir deps
Creating deps/foo.dep ...
mkdir objs
gcc -o objs/foo.o -c foo.c
ar crs ../../../build/libs/libfoo.a objs/foo.o

$ ls
Makefile  deps/  foo.c  objs/

$ ls ../../../build/libs/
libfoo.a

$ make clean
rm -fr objs deps  ../../../build/libs/libfoo.a

$ ls ../../../build/libs/
```

图 3.126

从运行的结果来看，的确是在 build/libs 目录下面生成了一个 libfoo.a 库文件。运行“make clean”以后，也并没有将 build/libs 这个目录删除，而只是删除了 libfoo.a 文件。

下面要做的是将这个 Makefile 运用到 code/huge/src 目录。最直接的想法是，将 foo 模块的 Makefile 拷贝到 huge/src 目录下面，然后做一些小的改动。但更好的方法是将各 Makefile 中公

共的部分抽取出来，通过复用的方式加以实现。

3.4.2 增进复用性

可以将公用部分放入一个独立的文件中——这就是 build 目录下 make.rule 文件的作用。那在 foo 模块的 Makefile 中，哪些是不能公用的呢？

（1）变量 EXE 和 LIB 的定义对于每一个软件模块是不同的。比如在 huge 项目中，需要将 code/foo/src 目录下 Makefile 中的 LIB 变量设置为“libfoo.a”，且 EXE 变量应当为空。但是，在 code/huge/src 目录中的 Makefile 内却要反过来，只定义 EXE 变量的值为“huge.exe”。

（2）DIR_EXES 变量和 DIR_LIBS 变量由于运用了相对路径，所以也是每个模块特有的。但是可以采用绝对路径的方式解决这个问题。比如，可以定义一个 ROOT 环境变量，其值设置为 huge 项目的根目录，这样的话，DIR_EXES 和 DIR_LIBS 就可以以 ROOT 为相对路径，从而使得其值对于所有的模块都相同。

在考虑复用的情形下，foo 模块的 Makefile 由两部分组成，分别是 build 目录中的 make.rule 和 code/foo/src 目录中的 Makefile，两部分内容如图 3.127 所示。

```
.PHONY: all clean

MKDIR = mkdir
RM = rm
RMFLAGS = -fr

CC = gcc
AR = ar
ARFLAGS = crs

DIR_OBJS = objs
DIR_EXES = $(ROOT)/build/exes
DIR_DEPS = deps
DIR_LIBS = $(ROOT)/build/libs
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS) $(DIR_LIBS)
RMS = $(DIR_OBJS) $(DIR_DEPS)

EXE =
ifneq ("$(EXE)", "")
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
RMS += $(EXE)
endif

LIB = libfoo.a
ifneq ("$(LIB)", "")
LIB := $(addprefix $(DIR_LIBS)/, $(LIB))
RMS += $(LIB)
endif

SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
```

```
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

ifeq ("$(wildcard $(DIR_OBJS))", "")
DEP_DIR_OBJS := $(DIR_OBJS)
endif
ifeq ("$(wildcard $(DIR_EXES))", "")
DEP_DIR_EXES := $(DIR_EXES)
endif
ifeq ("$(wildcard $(DIR_DEPS))", "")
DEP_DIR_DEPS := $(DIR_DEPS)
endif
ifeq ("$(wildcard $(DIR_LIBS))", "")
DEP_DIR_LIBS := $(DIR_LIBS)
endif

all: $(EXE) $(LIB)

ifneq ($(MAKECMDGOALS), clean)
include $(DEPS)
endif

$(DIRS):
	$(MKDIR) $@
$(EXE): $(DEP_DIR_EXES) $(OBJS)
	$(CC) -o $@ $(filter %.o, $^)
$(LIB): $(DEP_DIR_LIBS) $(OBJS)
	$(AR) $(ARFLAGS) $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DEP_DIR_OBJS) %.c
	$(CC) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DEP_DIR_DEPS) %.c
	@echo "Creating $@ ..."
	@set -e ; \
	$(RM) $(RMFLAGS) $@.tmp ; \
	$(CC) -E -MM $(filter %.c, $^) > $@.tmp ; \
	sed 's,\(.*\)\.o[ :]*,objs/\1.o $@: ,g' < $@.tmp > $@ ; \
	$(RM) $(RMFLAGS) $@.tmp
clean:
	$(RM) $(RMFLAGS) $(RMS)
```

```
EXE =

LIB = libfoo.a

include $(ROOT)/build/make.rule
```

图 3.127

foo 模块中的 Makfile 很简单，因为大部分内容都被移到了 make.rule 文件中。如果要运行 make，必须先在 Shell 上导出所需的 ROOT 变量，图 3.128 示例说明了操作步骤。

```
$ export ROOT=`pwd`

$ echo $ROOT
/Makefile/huge

$ cd code/foo/src
```

```
$ make
mkdir deps
Creating deps/foo.dep ...
mkdir objs
gcc -o objs/foo.o -c foo.c
ar crs /Makefile/huge/build/libs/libfoo.a objs/foo.o
                    /Makefile/huge/code/foo/src
$ make clean
rm -fr objs deps  /Makefile/huge/build/libs/libfoo.a
```

图 3.128

如果读者是一个 Linux 新手，需要注意，在导出 ROOT 变量时，除了先要进入 huge 项目的根目录外，pwd 命令前后的字符是“`”而不是“‘”。

接下来需要考虑 code/huge/src 目录中的 Makefile 了，我们希望在这个目录中存放的程序能生成一个可执行文件，在测试 Makefile 时，需要在目录中放置一个 main.c 文件，其中的内容如图 3.129 所示，而 Makefile 的内容如图 3.130 所示。

```
int main ()
{
   return 0;
}
```

图 3.129

```
EXE = huge.exe

LIB =

include $(ROOT)/build/make.rule
```

图 3.130

进入 code/huge/src 目录运行 make 检验 Makefile 是否能正常工作，结果如图 3.131 所示。从结果来看，huge.exe 文件被成功地生成了。

```
                    /Makefile/huge/code/huge/src
$ make
mkdir deps
Creating deps/main.dep ...
mkdir objs
gcc -o objs/main.o -c main.c
gcc -o /Makefile/huge/build/exes/huge.exe objs/main.o
                    /Makefile/huge/code/huge/src
$ ls $ROOT/build/exes
huge.exe
                    /Makefile/huge/code/huge/src
$ make clean
rm -fr objs deps /Makefile/huge/build/exes/huge.exe
```

图 3.131

3.4.3　支持头文件目录的指定

现在，是时候将项目文件放入各目录结构中了，图 3.132 示例说明了 huge 项目现有的三个文件。进入 code/foo/src 目录，运行 make 命令后的结果如图 3.133 所示。

```
#ifndef __FOO_H
#define __FOO_H

void foo ();

#endif
```

```
#include <stdio.h>
#include "foo.h"

void foo ()
{
   printf ("This is foo ()!\n");
}
```

```
#include "foo.h"

int main ()
{
   foo ();
   return 0;
}
```

图 3.132

```
/Makefile/huge/code/foo/src
$ make
/Makefile/huge/build/make.rule:50: deps/foo.dep: No such file or directory
mkdir deps
Creating deps/foo.dep ...
foo.c:2:17: error: foo.h: No such file or directory
make: *** [deps/foo.dep] Error 1
```

图 3.133

在构建 foo.dep 依赖文件时，gcc 因为找不到 foo.h 而报错并最终导致编译失败。根据图 3.123 所示的 huge 项目的目录结构，foo.c 和 foo.h 文件被分别存放在不同的目录中，在编译源文件时需要采用一定的形式告诉编译器到指定的目录中查找头文件，这得用到 gcc 的-I 选项。更改后的 Makefile 如图 3.134 所示。

```
.PHONY: all clean

MKDIR = mkdir
RM = rm
RMFLAGS = -fr

CC = gcc
```

```
AR = ar
ARFLAGS = crs

DIR_OBJS = objs
DIR_EXES = $(ROOT)/build/exes
DIR_DEPS = deps
DIR_LIBS = $(ROOT)/build/libs
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS) $(DIR_LIBS)
RMS = $(DIR_OBJS) $(DIR_DEPS)

ifneq ("$(EXE)", "")
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
RMS += $(EXE)
endif

ifneq ("$(LIB)", "")
LIB := $(addprefix $(DIR_LIBS)/, $(LIB))
RMS += $(LIB)
endif

SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

ifeq ("$(wildcard $(DIR_OBJS))", "")
DEP_DIR_OBJS := $(DIR_OBJS)
endif
ifeq ("$(wildcard $(DIR_EXES))", "")
DEP_DIR_EXES := $(DIR_EXES)
endif
ifeq ("$(wildcard $(DIR_DEPS))", "")
DEP_DIR_DEPS := $(DIR_DEPS)
endif
ifeq ("$(wildcard $(DIR_LIBS))", "")
DEP_DIR_LIBS := $(DIR_LIBS)
endif

all: $(EXE) $(LIB)

ifneq ($(MAKECMDGOALS), clean)
include $(DEPS)
endif

ifneq ($(INCLUDE_DIRS), "")
INCLUDE_DIRS := $(strip $(INCLUDE_DIRS))
INCLUDE_DIRS := $(addprefix -I, $(INCLUDE_DIRS))
endif

$(DIRS):
	$(MKDIR) $@
$(EXE): $(DEP_DIR_EXES) $(OBJS)
	$(CC) -o $@ $(filter %.o, $^)
$(LIB): $(DEP_DIR_LIBS) $(OBJS)
	$(AR) $(ARFLAGS) $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DEP_DIR_OBJS) %.c
	$(CC) $(INCLUDE_DIRS) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DEP_DIR_DEPS) %.c
	@echo "Creating $@ ..."
```

```
    @set -e ; \
    $(RM) $(RMFLAGS) $@.tmp ; \
    $(CC) $(INCLUDE_DIRS) -E -MM $(filter %.c, $^) > $@.tmp ; \
    sed 's,\(.*\)\.o[ :]*,objs/\1.o $@: ,g' < $@.tmp > $@ ; \
    $(RM) $(RMFLAGS) $@.tmp
clean:
    $(RM) $(RMFLAGS) $(RMS)
```

```
EXE =
LIB = libfoo.a

INCLUDE_DIRS = $(ROOT)/code/foo/inc

include $(ROOT)/build/make.rule
```

```
EXE = huge.exe
LIB =

INCLUDE_DIRS = $(ROOT)/code/foo/inc

include $(ROOT)/build/make.rule
```

图 3.134

make.rule 内的改动主要集中在增加了一个用于存放头文件目录的 INCLUDE_DIRS 变量上，这个变量可以根据需要存放任意数目的目录。为此，在 make.rule 中增加了一个条件语句块，该语句块在 INCLUDE_DIRS 变量的值不为空时，先采用 strip 函数去除多余的空格，然后再调用 addprefix 函数为 INCLUDE_DIRS 变量中的各目录名增加“-I”前缀。最后，在目标文件生成规则和依赖文件生成规则中增加了对 INCLUDE_DIRS 变量的引用，以便告诉 gcc 到哪里去找头文件。

code/foo/src 和 code/huge/src 目录下的 Makefile 所做的修改是为 INCLUDE_DIRS 变量设置正确的值。增加这些改动之后，编译 libfoo.a 的结果如图 3.135 所示。这一次 libfoo.a 被成功构建出来了。

```
                    /Makefile/huge/code/foo/src
$ make
/Makefile/huge/build/make.rule:43: deps/foo.dep: No such file or directory
mkdir deps
Creating deps/foo.dep ...
mkdir objs
gcc -I/Makefile/huge/code/foo/inc -o objs/foo.o -c foo.c
ar crs /Makefile/huge/build/libs/libfoo.a objs/foo.o

$ ls $ROOT/build/libs
libfoo.a
```

图 3.135

3.4.4　实现库链接

库已经有了，看看 huge 项目的可执行程序如何生成。进入 code/huge/src 目录，运行 make

的结果如图 3.136 所示。

```
                    /Makefile/huge/code/huge/src
$ make
/Makefile/huge/build/make.rule:43: deps/main.dep: No such file or directory
Making deps/main.dep ...
mkdir objs
gcc -I/Makefile/huge/code/foo/inc -o objs/main.o -c main.c
gcc -o /Makefile/huge/build/exes/huge.exe objs/main.o
objs/main.o:main.c:(.text+0x17): undefined reference to `foo'
collect2: ld returned 1 exit status
make: *** [/Makefile/huge/build/exes/huge.exe] Error 1
```

图 3.136

可以看到 main.o 目标文件被正确地生成了，但在链接时因为找不到 foo()函数的实现而出错了。由于 foo()函数的实现是放在 libfoo.a 库中的，而 Makefile 中并没有告诉编译器在生成 huge.exe 时需要与 libfoo.a 库一起链接。

解决这一问题的方法与前面指定头文件目录很相似，只是这一次要用到 gcc 的-l 和-L 选项（参见 4.3.7 节），更改后的 Makefile 如图 3.137 所示。

```
.PHONY: all clean

MKDIR = mkdir
RM = rm
RMFLAGS = -fr

CC = gcc
AR = ar
ARFLAGS = crs

DIR_OBJS = objs
DIR_EXES = $(ROOT)/build/exes
DIR_DEPS = deps
DIR_LIBS = $(ROOT)/build/libs
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS) $(DIR_LIBS)
RMS = $(DIR_OBJS) $(DIR_DEPS)

ifneq ("$(EXE)", "")
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
RMS += $(EXE)
endif

ifneq ("$(LIB)", "")
LIB := $(addprefix $(DIR_LIBS)/, $(LIB))
RMS += $(LIB)
endif

SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

ifeq ("$(wildcard $(DIR_OBJS))", "")
```

```
DEP_DIR_OBJS := $(DIR_OBJS)
endif
ifeq ("$(wildcard $(DIR_EXES))", "")
DEP_DIR_EXES := $(DIR_EXES)
endif
ifeq ("$(wildcard $(DIR_DEPS))", "")
DEP_DIR_DEPS := $(DIR_DEPS)
endif
ifeq ("$(wildcard $(DIR_LIBS))", "")
DEP_DIR_LIBS := $(DIR_LIBS)
endif

all: $(EXE) $(LIB)

ifneq ($(MAKECMDGOALS), clean)
include $(DEPS)
endif

ifneq ($(INCLUDE_DIRS), "")
INCLUDE_DIRS := $(strip $(INCLUDE_DIRS))
INCLUDE_DIRS := $(addprefix -I, $(INCLUDE_DIRS))
endif
ifneq ($(LINK_LIBS), "")
LINK_LIBS := $(strip $(LINK_LIBS))
LINK_LIBS := $(addprefix -l, $(LINK_LIBS))
endif

$(DIRS):
	$(MKDIR) $@
$(EXE): $(DEP_DIR_EXES) $(OBJS)
	$(CC) -L$(DIR_LIBS) -o $@ $(filter %.o, $^) $(LINK_LIBS)
$(LIB): $(DEP_DIR_LIBS) $(OBJS)
	$(AR) $(ARFLAGS) $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DEP_DIR_OBJS) %.c
	$(CC) $(INCLUDE_DIRS) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DEP_DIR_DEPS) %.c
	@echo "Creating $@ ..."
	@set -e ; \
	$(RM) $(RMFLAGS) $@.tmp ; \
	$(CC) $(INCLUDE_DIRS) -E -MM $(filter %.c, $^) > $@.tmp ; \
	sed 's,\(.*\)\.o[ :]*,objs/\1.o $@: ,g' < $@.tmp > $@ ; \
	$(RM) $(RMFLAGS) $@.tmp
clean:
	$(RM) $(RMFLAGS) $(RMS)
```

```
EXE =
LIB = libfoo.a

INCLUDE_DIRS = $(ROOT)/code/foo/inc
LINK_LIBS =

include $(ROOT)/build/make.rule
```

```
EXE = huge.exe
LIB =

INCLUDE_DIRS = $(ROOT)/code/foo/inc
LINK_LIBS = foo

include $(ROOT)/build/make.rule
```

图 3.137

其中的改动有：

（1）在 make.rule 文件中增加了对 LINK_LIBS 变量的引用，这个变量用来存放可执行程序在链接时所需要用到的所有库。

（2）在 make.rule 中将$（DIR_LIBS）通过 gcc 的-L 选项加入到了编译器的库搜索目录列表中。在 huge 项目中，采用将所有的库文件都放入$(DIR_LIBS)目录中这种方式简化了 Makefile 的实现。

（3）在各模块的 src 目录下的 Makefile 中增加了 LINK_LIBS 变量的定义，且在 code/huge/src/Makefile 中对 LINK_LIBS 赋值为“foo”。在 Linux 中，一个静态库名的格式为 libXXX.a，其中的 XXX 就是采用 gcc 的-l 选项时所需给的名。在这个 Makefile 中，LINK_LIBS 变量的“foo”值就是表示指定 libfoo.a 库。

图 3.138 是新编译系统下编译 huge.exe 可执行程序的结果。这一次正确地生成了可执行文件，图中还示例说明了它的执行结果。

```
                    /Makefile/huge/code/huge/src
$ make
/Makefile/huge/build/make.rule:43: deps/main.dep: No such file or directory
mkdir deps
Creating deps/main.dep ...
mkdir objs
gcc -I/Makefile/huge/code/foo/inc -o objs/main.o -c main.c
gcc -L/Makefile/huge/build/libs -o /Makefile/huge/build/exes/huge.exe objs/main.o -lfoo
                    /Makefile/huge/code/huge/src
$ $ROOT/build/exes/huge.exe
This is foo ()!
```

图 3.138

现在，假设想往 huge 项目中增加一个 bar 模块，这个模块将生成 libbar.a 静态库。我们可以通过增加新模块的方式来检验 huge 项目的编译系统设计得如何。bar 模块的源程序及 Makefile 如图 3.139 所示。

```
#ifndef __BAR_H
#define __BAR_H

void bar ();

#endif
```

```
#include <stdio.h>
#include "bar.h"

void bar ()
{
   printf ("This is bar ()!\n");
}
```

```
EXE =
LIB = libbar.a

INCLUDE_DIRS = $(ROOT)/code/bar/inc
LINK_LIBS =

include $(ROOT)/build/make.rule
```

图 3.139

在图 3.123 所示的目录结构基础上，增加如图 3.140 所示的目录结构用于存放 bar 模块的源程序。

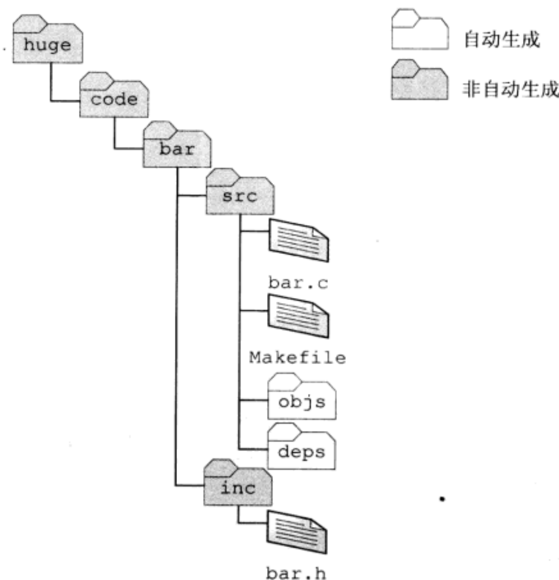


图 3.140

为了构建 libbar.a，需要进入 code/bar/src 目录并运行 make 命令，结果如图 3.141 所示。

```
                /Makefile/huge/code/bar/src
$ make
/Makefile/huge/build/make.rule:43: deps/bar.dep: No such file or directory
mkdir deps
Creating deps/bar.dep ...
mkdir objs
gcc -I/Makefile/huge/code/bar/inc -o objs/bar.o -c bar.c
ar crs /Makefile/huge/build/libs/libbar.a objs/bar.o
                /Makefile/huge/code/bar/src
$ ls $ROOT/build/libs
libbar.a  libfoo.a
```

图 3.141

很明显，当需要增加一个软件模块时，在 Makefile 方面需要做的工作非常少。只需将已存在的一个 Makefile 拷贝过来，然后进行一些很小的修改就行了，这是好的编译系统应当具备的

特点。如果 huge.exe 需要使用 libbar.a 库，则需要进行图 3.142 所示的修改。

```
#include "foo.h"
#include "bar.h"

int main ()
{
    foo ();
    bar ();
    return 0;
}
```

```
EXE = huge.exe
LIB =

INCLUDE_DIRS = $(ROOT)/code/foo/inc \
    $(ROOT)/code/foo/inc

LINK_LIBS = foo bar

include $(ROOT)/build/make.rule
```

图 3.142

更改体现在 INCLUDE_DIRS 和 LINK_LIBS 变量上，一个用于指定头文件目录，另一个用于指定链接 libbar.a 库。图 3.143 示例说明了 huge.exe 可执行程序的编译和运行结果。

```
                    /Makefile/huge/code/huge/src
$ make
Creating deps/main.dep ...
gcc -I/Makefile/huge/code/foo/inc -I/Makefile/huge/code/bar/inc -o objs/main.o -c main.c
gcc -L/Makefile/huge/build/libs -o /Makefile/huge/build/exes/huge.exe objs/main.o -lfoo
-lbar
                    /Makefile/huge/code/huge/src
$ $ROOT/build/exes/huge.exe
This is foo ()!
This is bar ()!
```

图 3.143

3.4.5　增强可使用性

从前面看来，为了编译 huge 项目需要进入不同的目录运行 make，我们可以简化这一活动，这得通过图 3.123 中所规划的 build 目录下面的 Makefile 来实现。

这一次不打算逐步地介绍这个 Makefile 是如何设计出来的的，而是直接列于图 3.144 中。以读者目前所掌握的知识可以很容易地理解它。

```
.PHONY: all clean

ROOT = $(realpath ..)
```

```
DIRS = $(ROOT)/code/foo/src \
    $(ROOT)/code/bar/src \
    $(ROOT)/code/huge/src

RM = rm
RMFLAGS = -fr
RMS = $(ROOT)/build/exes $(ROOT)/build/libs

all clean:
    @set -e; \
    for dir in $(DIRS); \
    do \
        cd $$dir && $(MAKE) ROOT=$(ROOT) $@; \
    done
    @set -e; \
    if [ "$(MAKECMDGOALS)" = "clean" ]; then $(RM) $(RMFLAGS) $(RMS); fi
    @echo ""
    @echo ":-) Completed"
    @echo ""
```

图 3.144

在这个 Makefile 中，使用了 realpath 函数以获得项目的根目录，并将这个值传递给每一个 DIRS 变量中所列出目录中的 Makefile。这样做的好处是，如果从 build 目录下编译整个项目，那么并不需要在 Shell 上导出 ROOT 环境变量。

另外，这个 Makefile 还使用了 Shell 中的 for 语句，用于遍历 DIRS 变量中的每一个目录，以便在各目录中运行 make 命令。另外，还用到了在 3.2.2.2 节的“2. 特殊变量”小节中提到的 MAKE 变量。

最后，由于库必须比可执行程序先构建出来，所以在 DIRS 变量中必须将库目录放在可执行程序目录之前。

图 3.145 所示的运行结果证明，这个 Makefile 的确简化了项目的编译工作。当然，我们仍可以采用前面单独构建的方式，只是之前必须保证 ROOT 环境变量被正确地导出了，这一点务必牢记。

```
                    /Makefile/huge/build
$ make
make[1]: Entering directory `/Makefile/huge/code/foo/src'
/Makefile/huge/build/make.rule:43: deps/foo.dep: No such file or directory
mkdir deps
Creating deps/foo.dep ...
make[1]: Leaving directory `/Makefile/huge/code/foo/src'
make[1]: Entering directory `/Makefile/huge/code/foo/src'
mkdir /Makefile/huge/build/libs
mkdir objs
gcc -I/Makefile/huge/code/foo/inc -o objs/foo.o -c foo.c
ar crs /Makefile/huge/build/libs/libfoo.a objs/foo.o
make[1]: Leaving directory `/Makefile/huge/code/foo/src'
make[1]: Entering directory `/Makefile/huge/code/bar/src'
/Makefile/huge/build/make.rule:43: deps/bar.dep: No such file or directory
mkdir deps
Creating deps/bar.dep ...
```

```
make[1]: Leaving directory `/Makefile/huge/code/bar/src'
make[1]: Entering directory `/Makefile/huge/code/bar/src'
mkdir objs
gcc -I/Makefile/huge/code/bar/inc -o objs/bar.o -c bar.c
ar crs /Makefile/huge/build/libs/libbar.a objs/bar.o
make[1]: Leaving directory `/Makefile/huge/code/bar/src'
make[1]: Entering directory `/Makefile/huge/code/huge/src'
/Makefile/huge/build/make.rule:43: deps/main.dep: No such file or directory
mkdir deps
Creating deps/main.dep ...
make[1]: Leaving directory `/Makefile/huge/code/huge/src'
make[1]: Entering directory `/Makefile/huge/code/huge/src'
mkdir /Makefile/huge/build/exes
mkdir objs
gcc -I/Makefile/huge/code/foo/inc -I/Makefile/huge/code/bar/inc -o
    objs/main.o -c main.c
gcc -L/Makefile/huge/build/libs -o /Makefile/huge/build/exes/huge.exe
    objs/main.o -lfoo -lbar
make[1]: Leaving directory `/Makefile/huge/code/huge/src'

:-) Completed

                    /Makefile/huge/build
$ make clean
make[1]: Entering directory `/Makefile/huge/code/foo/src'
rm -fr objs deps /Makefile/huge/build/libs/libfoo.a
make[1]: Leaving directory `/Makefile/huge/code/foo/src'
make[1]: Entering directory `/Makefile/huge/code/bar/src'
rm -fr objs deps /Makefile/huge/build/libs/libbar.a
make[1]: Leaving directory `/Makefile/huge/code/bar/src'
make[1]: Entering directory `/Makefile/huge/code/huge/src'
rm -fr objs deps /Makefile/huge/build/exes/huge.exe
make[1]: Leaving directory `/Makefile/huge/code/huge/src'
rm -fr /Makefile/huge/build/exes /Makefile/huge/build/libs

:-) Completed
```

图 3.145

在 build 目录下进行项目编译时，只要在构建的最后看到一张笑脸出现在终端上，那就意味着项目编译成功了。

3.4.6　管理对库的依赖关系

3.3.2 节就使用 Makefile 进行源程序之间的依赖关系管理进行了探讨，在 complicated 项目的最后也设计出了一个能应对所有源程序依赖关系的编译系统。huge 项目的编译系统最大的变化是增加了对库的支持，也正是因为库的出现使得依赖关系变得更加复杂。

为了发现因为库的引入而产生的依赖关系问题，需要在 huge 项目上做一个小试验，这个试验是模拟对 foo 模块内 foo.c 文件的修改，然后检查整个项目的编译情况，结果如图 3.146 所示。

```
$ touch /Makefile/huge/code/foo/src/foo.c
```

```
$ make
make[1]: Entering directory `/Makefile/huge/code/foo/src'
Creating deps/foo.dep ...
make[1]: Leaving directory `/Makefile/huge/code/foo/src'
make[1]: Entering directory `/Makefile/huge/code/foo/src'
gcc -I/Makefile/huge/code/foo/inc -o objs/foo.o -c foo.c
ar crs /Makefile/huge/build/libs/libfoo.a objs/foo.o
make[1]: Leaving directory `/Makefile/huge/code/foo/src'
make[1]: Entering directory `/Makefile/huge/code/bar/src'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/Makefile/huge/code/bar/src'
make[1]: Entering directory `/Makefile/huge/code/huge/src'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/Makefile/huge/code/huge/src'

:-) Completed
```

图 3.146

从结果来看，foo.c 文件的更改导致了 libfoo.a 库文件被重新编译，这是所期望的行为。但是，libfoo.a 的更改应当进一步使得 huge.exe 也被重新编译，这一点从结果来看并没发生。出现这一现象的原因读者或许很快就反应过来了，因为 Makefile 中并没有反映出 huge.exe 对于 libfoo.a 库的依赖关系，因此 make 无法知道当 libfoo.a 被更新的情形下应对 huge.exe 进行重新构建。

解决问题的思路还是很清晰的。回顾图 3.142 中的 Makefile，其中 LINK_LIBS 已经指明了 huge.exe 生成时所需使用的库，我们也可以让 huge.exe 依赖于这些库。要让 huge.exe 依赖于这些库，则必须列出库的具体路径，这一点容易做到，因为 huge 项目所生成的库都存放在 build/libs 目录中。读者可能会想到，LINK_LIBS 变量中可能会指定一些并非是由 huge 项目生成的库，比如系统库 libc.a 等。显然，对于不是由 huge 项目生成的库，huge.exe 不应当依赖于它们。为此，对于在 LINK_LIBS 变量中列出的库，应当检查它们是否存在于 build/libs 目录中，如果不存在于这一目录中，则不应将这些库当做是 huge.exe 的先决条件。图 3.147 列出了对 make.rule 文件的变更。

```
.PHONY: all clean

MKDIR = mkdir
RM = rm
RMFLAGS = -fr

CC = gcc
AR = ar
ARFLAGS = crs

DIR_OBJS = objs
DIR_EXES = $(ROOT)/build/exes
DIR_DEPS = deps
DIR_LIBS = $(ROOT)/build/libs
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS) $(DIR_LIBS)
RMS = $(DIR_OBJS) $(DIR_DEPS)

ifneq ("$(EXE)", "")
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
```

```
RMS += $(EXE)
endif

ifneq ("$(LIB)", "")
LIB := $(addprefix $(DIR_LIBS)/, $(LIB))
RMS += $(LIB)
endif

SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

ifeq ("$(wildcard $(DIR_OBJS))", "")
DEP_DIR_OBJS := $(DIR_OBJS)
endif
ifeq ("$(wildcard $(DIR_EXES))", "")
DEP_DIR_EXES := $(DIR_EXES)
endif
ifeq ("$(wildcard $(DIR_DEPS))", "")
DEP_DIR_DEPS := $(DIR_DEPS)
endif
ifeq ("$(wildcard $(DIR_LIBS))", "")
DEP_DIR_LIBS := $(DIR_LIBS)
endif

all: $(EXE) $(LIB)

ifneq ($(MAKECMDGOALS), clean)
include $(DEPS)
endif

ifneq ($(INCLUDE_DIRS), "")
INCLUDE_DIRS := $(strip $(INCLUDE_DIRS))
INCLUDE_DIRS := $(addprefix -I, $(INCLUDE_DIRS))
endif
ifneq ($(LINK_LIBS), "")
LINK_LIBS := $(strip $(LINK_LIBS))
LIB_ALL := $(notdir $(wildcard $(DIR_LIBS)/*))
LIB_FILTERED := $(addsuffix %, $(addprefix lib, $(LINK_LIBS)))
$(eval DEP_LIBS = $(filter $(LIB_FILTERED), $(LIB_ALL)))
DEP_LIBS := $(addprefix $(DIR_LIBS)/, $(DEP_LIBS))
LINK_LIBS := $(addprefix -l, $(LINK_LIBS))
endif

$(DIRS):
	$(MKDIR) $@
$(EXE): $(DEP_DIR_EXES) $(OBJS) $(DEP_LIBS)
	$(CC) -L$(DIR_LIBS) -o $@ $(filter %.o, $^) $(LINK_LIBS)
$(LIB): $(DEP_DIR_LIBS) $(OBJS)
	$(AR) $(ARFLAGS) $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DEP_DIR_OBJS) %.c
	$(CC) $(INCLUDE_DIRS) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DEP_DIR_DEPS) %.c
	@echo "Creating $@ ..."
	@set -e ; \
	$(RM) $(RMFLAGS) $@.tmp ; \
	$(CC) $(INCLUDE_DIRS) -E -MM $(filter %.c, $^) > $@.tmp ; \
	sed 's,\(.*\)\.o[ :]*,objs/\1.o $@: ,g' < $@.tmp > $@ ; \
```

```
    $(RM) $(RMFLAGS) $@.tmp
clean:
    $(RM) $(RMFLAGS) $(RMS)
```

图 3.147

其中新增的部分第一次使用了 notdir 和 eval 函数以获得 huge.exe 所依赖的库文件的具体路径。$（DEP_LIBS）最后被当做$（EXE）目标的先决条件。图 3.148 所示为更改后的运行结果。可以发现，这次 huge.exe 也因为 foo.c 文件的更改而被重新构建了。

```
$ touch /Makefile/huge/code/foo/src/foo.c

$ make
make[1]: Entering directory `/Makefile/huge/code/foo/src'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/Makefile/huge/code/foo/src'
make[1]: Entering directory `/Makefile/huge/code/bar/src'
Creating deps/bar.dep ...
make[1]: Leaving directory `/Makefile/huge/code/bar/src'
make[1]: Entering directory `/Makefile/huge/code/bar/src'
gcc -I/Makefile/huge/code/bar/inc -o objs/bar.o -c bar.c
ar crs /Makefile/huge/build/libs/libbar.a objs/bar.o
make[1]: Leaving directory `/Makefile/huge/code/bar/src'
make[1]: Entering directory `/Makefile/huge/code/huge/src'
gcc -L/Makefile/huge/build/libs -o /Makefile/huge/build/exes/huge.exe
    objs/main.o -lfoo -lbar
make[1]: Leaving directory `/Makefile/huge/code/huge/src'

:-) Completed
```

图 3.148

3.4.7 改善编译效率

对于大型项目，提高项目的编译速度有着非常重要的意义，因为它意味着更具可开发性（参见第 18 章）。虽然在本章我们已为项目建立了有效的依赖关系，使得每一次项目编译只需编译修改的部分。但是，获得高效的编译系统，还需通过其他的努力。

从 Makefile 的角度来看，一个可以改善编译效率的地方与其中的隐式规则有关。为了了解 make 的隐式规则，可以对来自 simple 项目的 Makefile（图 3.58）做一点改动，即删除生成.o 文件的规则（与隐式规则相对应的是，在 Makefile 中定义的规则称为显式规则），如图 3.149 所示。

```
.PHONY: clean

CC = gcc
RM = rm

EXE = simple.exe
SRCS = $(wildcard *.c)
OBJS = $(patsubst %.c,%.o,$(SRCS))
```

```
$(EXE) : $(OBJS)
    $(CC) -o $@ $^
%.o: %.c
    $(CC) -o $@ -c $^
clean:
    $(RM) -fr $(EXE) $(OBJS)
```

图 3.149

接着运行 make 以编译 simple 项目，其结果如图 3.150 所示。可见，尽管在 Makefile 中删除了生成.o 文件的规则，但是 make 还是能成功生成相应的.o 文件，这正是因为 make 存在自带隐式规则的缘故。

```
                    /Makefile/simple
$ make
gcc -c -o foo.o foo.c
gcc -c -o main.o main.c
gcc -o simple.exe foo.o main.o
                    /Makefile/simple
$ make clean
rm -fr simple.exe foo.o main.o
```

图 3.150

在 make 中存在大量的隐式规则，通过隐式规则将大大简化 Makefile 的编写[1]。这里简化后的 Makefile 之所以能工作，是因为 make 中存在图 3.151 所示的隐式规则。

```
%.o: %.c
    $(CC) -c $(CPPFLAGS) $(CFLAGS) $^
```

图 3.151

从图 3.151 所示的隐式规则可以看出，如果想将生成的.o 文件放入特定的目录中，那么它就显得无能为力了，因为它没有使用编译器的-o 选项以指明生成文件的位置。另外，隐式规则的使用对于大型项目存在一个副作用，因为 make 需要查找隐式规则而降低编译效率。为了禁用 make 所自带的隐式规则，可以通过使用 make 的-r 选项来实现。图 3.152 显示了在使用-r 选项的情形下，将导致简化后的 Makefile 无法用于成功编译项目。

```
                    /Makefile/simple
$ make -r
make: *** No rule to make target `foo.o', needed by `simple.exe'.  Stop.
```

图 3.152

隐式规则是可以被覆盖的。当在 simple 项目中定义了生成.o 文件的规则时，make 就以它作为最终生成.o 文件的规则，因为该规则覆盖了 make 自带的隐式规则。

我们知道，本章所设计出的编译系统，都没有使用到 make 自带的隐式规则。但 make 却

① 在 make 的官方使用手册《GNU Make》中可以找到 make 所支持的所有隐式规则。该手册在附书光盘中能找到，其文件名为 make.pdf。

并不清楚这一点，它还是会在读入 Makefile 以后从其自带的隐式规则中开始查找，看是否能找到用于编译源程序的规则。如果一个项目没有采用隐式规则，那么，最好告诉 make，“请不要为我的项目查找隐式规则，否则太浪费时间了！”

我们可以运用 make 的-r 选项来提高项目的编译效率，更改后的 Makefile 如图 3.153 所示。通过结合使用 make 的-d 选项（参见 3.6 节）可以验证，-r 选项的使用能提高每个源文件的编译效率。

huge/build/Makefile

```
.PHONY: all clean

ROOT = $(realpath ..)

DIRS = $(ROOT)/code/foo/src \
   $(ROOT)/code/bar/src \
   $(ROOT)/code/huge/src

RM = rm
RMFLAGS = -fr
RMS = $(ROOT)/build/exes $(ROOT)/build/libs

all clean:
   @set -e; \
   for dir in $(DIRS); \
   do \
       cd $$dir && $(MAKE) -r ROOT=$(ROOT) $@; \
   done
   @set -e; \
   if [ "$(MAKECMDGOALS)" == "clean" ] ; then $(RM) $(RMFLAGS) $(RMS) ; fi
   @echo ""
   @echo ":-) Completed"
   @echo ""
```

图 3.153

3.4.8　恰当地书写注释

读者可能已发现，本章的 Makefile 中找不到任何注释。之所以出现这种情形，不是因为 Makefile 中不能写注释，而是因为作者想节省书的篇幅。一个具有较好可维护性的编译系统，在 Makefile 中提供适当的注释是很有必要的。另外，13.6.5 节中所提出的通过命名传达设计意图这一设计原则，同样适用于 Makefile 的设计。

图 3.154 示例说明了在 Makefile 中如何加入注释。请注意，注释是以“#”开始的，它既可以占用完整的一行（如图中内容的第一行），也可以放在一行的后面（如图中内容的第二行）。

```
# This is a comment line
all: # This is another comment line
   echo "Hello World"
```

图 3.154

3.5 理解 make 的解析行为

make 是以从上到下的顺序读入 Makefile 中的内容的。然而，处理 Makefile 中的语句却并非完全从上到下。

大体上，make 处理一个 Makefile 分两个阶段。第一个阶段包含：

（1）make 读入 Makefile，以及 Makefile 中所包含的其他 Makefile。

（2）make 分析并获得变量名、变量值、隐式规则和显式规则。

（3）构建所有目标的关系树，以及它们的先决条件。

在第二个阶段，make 基于第一个阶段所建立的内部结构分析哪些目标需要重新构建，以及需要执行哪些规则的命令来构建这些目标。

理解 make 处理 Makefile 的两个阶段对于熟练地编写 Makefile 非常重要。就作者的学习经验来看，也曾经因为不了解这两个阶段而产生过不少困惑。

make 在处理 Makefile 中的语句时，存在立即展开和延迟展开两种类别。立即展开是指语句在第一个处理阶段就被展开。变量和函数就是立即展开的。延迟展开则是指当 make 处理它时并不立即展开，其展开动作发生在第二个阶段。

对于不同的语句，make 将采用不同的展开策略。图 3.155 示例说明了与赋值相关的展开策略。注意，左边的变量名总是立即展开的，而右边的变量值却未必。其中“+=”的左边有可能采用立即展开也有可能采用延迟展开。当左边的变量名在使用“+=”之前如已被设置为简单扩展变量（即采用“:=”赋值）时，则采用立即展开的方式，否则采用延迟展开的方式。

```
立即展开 = 延迟展开
立即展开 ?= 延迟展开
立即展开 := 立即展开
立即展开 += 立即展开或延迟展开
```

图 3.155

对于所有的条件语句，make 都采用立即展开的方式，这包括 ifdef、ifeq、ifndef 和 ifneq。因为自动变量都是在规则所对应的命令块中被设置的，而这一过程发生在第二个阶段，即自动变量是延迟展开的，所以自动变量在条件语法中不能使用。如果非得在条件语法中使用条件变量，则只能使用 Shell 的条件语法，而不是 make 的。

对于所有规则的展开策略，如图 3.156 所示。自动变量由于只能存在于规则的命令部分，所以一定是延迟展开的。

```
立即展开 : 立即展开
    延迟展开
```

图 3.156

3.6 Makefile 的调试

Makefile 的调试是相对困难的事，因为不存在断点设置等高级编程语言所具有的调试手段。就作者的经验来看，调试 Makefile 主要有两种方法。

第一种方法是通过 make 的-d 和--debug 两个参数来查看 make 是如何处理 Makefile 的。通过使用这两个参数，可以帮助分析 Makefile 的构建行为为什么不是预期的那样。

在没有使用-r 选项的情况下，通过使用-d 参数可以看到 make 是如何匹配隐式规则的。这里不打算列出使用-d 参数的效果，读者可以自行试一试以了解 make 的一些幕后行为。

--debug 参数则更灵活，在使用它时，还得加上另外的参数值，比如"make --debug=a"与"make -d"的效果是完全一样的。图 3.157 示例说明了--debug 可以使用的参数值及其含义。

参数值	含 义
a	列出所有的调试信息，--debug=a 与-d 的效果完全一样
b	打印出基本的调试信息
v	输出比使用 b 参数值更详细的调试信息
i	显示隐式规则
j	展示命令调用的细节
m	用于重新构建 Makefile 时的调试

图 3.157

读者如果试过了以上命令参数将发现，Makefile 中的变量及其值并不会被输出。而调试 Makefile 的过程中，在很多情况下我们需要了解变量的值，这就得使用第二种方法了。

第二种方法使用 Shell 的 echo 命令。需要注意一点，由于 echo 是命令，它不能在规则命令体之外被调用。因此，图 3.158 试图输出 ROOT 变量值的 Makefile 将获得来自 make 的错误报告。取而代之的是，图 3.159 就能达到查看 ROOT 变量的值这一目的。

```
ROOT := $(abspath /usr/../lib)
echo $(ROOT)
```

图 3.158

```
.PHONY: all

ROOT := $(abspath /usr/../lib)

all:
    @echo $(ROOT)
```

图 3.159

使用这种调试方法需要先找到合适的 echo 命令放置点，所以对 Makefile 的实现要非常熟悉——要完全理解 Makefile 所描述的依赖关系树，以及 make 的工作机理。

3.7 make 的常用选项

本章涉及了 make 命令的多个选项，这里总结一下 make 的常用选项，如图 3.160 所示。对于每一个选项，只作简单的说明性介绍，读者可以通过在自己的学习环境中运行 make 命令加上相应的选项，以了解各选项的确切含义。

命令参数	含　义
-d	显示调试信息，参见 3.6 节
--debug	显示调试信息，参见 3.6 节
-h	示例说明 make 简单的命令参数帮助信息
-k	忽略构建过程中的错误
-n	在构建过程中，只打印出命令，而不真正地调用这些命令
-p	列出 make 的数据库，其中包含了隐式规则的定义等
-r	忽略 make 的自带隐式规则
-s	采用静默模式，不打印 make 在构建过程中所调用的命令
-t	用 touch 命令更改目标的时间戳，而不对目标进行重新构建
-v	显示 make 的版本

图 3.160

3.8 活用 make

make 可以被运用在很多需要有规律地批量处理文件的场合，这一节将介绍活用它的两个例子。

如果希望 make 能正常工作，则各源文件的时间戳非常重要，它们必须不能比计算机上的时间更新，否则 make 会告警并指出项目编译可能不完整。为了模拟这类问题，读者可以先对 huge 项目进行 make，然后将计算机的时间更改为慢一点，再运行一次 make。在这种情形下将获得图 3.161 所示那样的警告。在现实中，这种问题的出现通常不是因为计算机的时间更改了，而是因为将整个项目从一个计算机拷贝到另一个计算机中，且两个计算机的时间存在不一致。

```
/Makefile/huge/build
$ make
make[1]: Entering directory `/Makefile/huge/code/foo/src'
make[1]: Warning: File `deps/foo.dep' has modification time 32 s in the future
make[1]: Nothing to be done for `all'.
make[1]: warning:  Clock skew detected.  Your build may be incomplete.
make[1]: Leaving directory `/Makefile/huge/code/foo/src'
make[1]: Entering directory `/Makefile/huge/code/bar/src'
```

```
make[1]: Warning: File `deps/bar.dep' has modification time 36 s in the future
make[1]: Nothing to be done for `all'.
make[1]: warning:  Clock skew detected.  Your build may be incomplete.
make[1]: Leaving directory `/Makefile/huge/code/bar/src'
make[1]: Entering directory `/Makefile/huge/code/huge/src'
make[1]: Warning: File `deps/main.dep' has modification time 40 s in the future
make[1]: Nothing to be done for `all'.
make[1]: warning:  Clock skew detected.  Your build may be incomplete.
make[1]: Leaving directory `/Makefile/huge/code/huge/src'

:-) Completed
```

图 3.161

解决这类问题的方法，可以使用 touch 命令将每一个文件的时间更改为与计算机同步，当然只使用 touch 命令对每一个文件进行更改不是一件易事，还得使用 find 命令。图 3.162 示例说明了如何使用两个命令，更改项目所有文件的时间戳。

```
                    /Makefile/huge/build
$ find ../ -exec touch {} \;
```

图 3.162

对于不熟悉这一方法的读者来说，记住这两个命令的组合用法可能比较麻烦，更好的解决方法是在 Makefile 中实现一个 touch 目标，如图 3.163 所示。有了这样的 Makefile，当需要同步文件时间戳时，只要运行“make touch”就行了。

```
.PHONY: all clean touch

ROOT = $(realpath ..)

DIRS = $(ROOT)/code/foo/src \
    $(ROOT)/code/bar/src \
    $(ROOT)/code/huge/src

RM = rm
RMFLAGS = -fr
RMS = $(ROOT)/build/exes $(ROOT)/build/libs

all clean:
    @set -e; \
    for dir in $(DIRS); \
    do \
        cd $$dir && $(MAKE) -r ROOT=$(ROOT) $@; \
    done
    @set -e; \
    if [ "$(MAKECMDGOALS)" == "clean" ] ; then $(RM) $(RMFLAGS) $(RMS) ; fi
    @echo ""
    @echo ":-) Completed"
    @echo ""

touch:
    @echo "Processing ..."
    @find $(ROOT) -exec touch {} \;
    @echo ""
    @echo ":-) Completed"
    @echo ""
```

图 3.163

下面再看一个使用 make 进行文件重命名的例子。在有些情形下，可能需要将某目录下的大量文件进行有规律的重命名。处理这类事务正是 make 的强项，图 3.164 示例说明了 backup 和 restore 两个 Makefile。backup 用于将.c 文件全部重命名为.c.bak 文件，而 restore 则反过来。图 3.165 测试了这两个 Makefile 的功能。

```
FILES = $(wildcard *.c)
FILES := $(FILES:.c=.c.bak)

backup: $(FILES)

%.c.bak: %.c
   mv $^ $@
```

```
FILES = $(wildcard *.c.bak)
FILES := $(FILES:.c.bak=.c)

backup: $(FILES)s

%.c: %.c.bak
   mv $^ $@
```

图 3.164

```
$ ls
backup  bar.c  foo.c  restore

$ make -f backup
mv bar.c bar.c.bak
mv foo.c foo.c.bak

$ ls
backup  bar.c.bak  foo.c.bak  restore

$ make -f restore
mv bar.c.bak bar.c
mv foo.c.bak foo.c

$ ls
backup  bar.c  foo.c  restore
```

图 3.165

除了这里讲解的两个例子外，在本书的质量保证篇读者还将看到如何使用 Makefile 将其他的工具无缝地整合到开发环境中。到时候，相信读者也会认同 make 是项目的全能管家。

3.9　小结

本章一开始以“Hello World”这一最简单的 Makefile 为例介绍了规则，并通过做 simple、complicated 和 huge 三个虚拟项目，系统性地介绍了 Makefile 中的知识点，以及详细地说明了如何设计一个高效和专业的编译系统。在本章的末尾，也阐述了如何对 Makefile 进行调试，以

及揭示了 make 的幕后行为。

尽管 huge 项目的编译系统比较完备了，但它离真实项目的编译环境还是有些差距的，主要体现在灵活性不足。比如，可能不同的库需要采用不同的编译选项进行编译，等等。

从质量保证篇的第 28 章开始，会介绍位于光盘中 Project/embedded 目录下的编译环境是如何设计的。相比之下，光盘中 embedded 目录下的编译系统是个完全可以被运用到真实项目的编译环境。读者可以放心的是，构建 embedded 项目的编译系统所需的知识全在本章介绍了。



第 4 章 gcc，C语言编译器

可以说，嵌入式软件开发所使用的编译工具是 GNU（http://www.gnu.org）的天下。这个改变了软件产业格局的组织除了为我们带来了免费的 Linux 操作系统外，更带来了很多其他开源软件项目，GCC 编译器就位列其中。读者可能会问："哪些嵌入式软件平台是采用 GCC 编译器的呢？"

首先，采用 Linux 内核的实时或非实时嵌入式系统的开发就不用说了，它们全都采用 GCC[①]；另一个很有名的来自 WindRiver（现已被 Intel 收购）的 VxWorks 操作系统，也可以采用 GCC 进行编译[②]；还有就是 RTEMS 和 eCos 这两个开源的操作系统也是采用 GCC 的。如果要列举出所有采用 GCC 编译器进行编译的操作系统的话，相信这个列表会很长。

GCC 是"GNU Compiler Collection"的缩写，从字面意思来看它是一个"编译器集"。是的，GCC 不光能用于编译 C/C++语言程序，还能用于编译 Java、Objective-C 等语言程序。在本章中，我们只关注 GCC 的 C 语言编译功能，所以只要说到 gcc 就是默指 C 语言编译器，且用小写字母的形式特指。

本章并不是从零开始向读者介绍如何使用 gcc 来编译程序，而是假设读者知道如何使用它。也不打算介绍 gcc 的所有编译选项，而是着重介绍实用但读者很有可能并不了解的部分选项。

4.1 什么是交叉编译器

在嵌入式系统开发中，经常用到与交叉编译（cross compiling）相关的术语，比如，交叉编译器、交叉链接器、交叉编译环境等。那交叉编译到底是指什么呢？

嵌入式产品的资源往往很有限，如果它的资源和我们平常用的桌面计算机（后面称为开发主机或简称为主机，host machine）一样，那就不存在交叉编译一说了。最为典型的是，嵌入式系统的内存往往是几十兆字节，且只有闪存而没有硬盘这种大容量存储设备。在这种资源有限

---

① Linux 非实时操作系统包括 RedHat Linux、Fedora、Ubuntu、CentOS 等；Linux 实时操作系统包括 MontaVista Linux、WindRiver Linux、RTLinux 等。

② VxWorks 除了支持采用 GCC 编译器进行编译外，还支持来自 WindRiver 公司自己的 Diab 编译器。

的环境中，不可能将开发工具安装在嵌入式设备中（后面称为目标机，target machine），然后像平时做桌面软件开发那样在嵌入式设备上直接进行软件开发。因此，嵌入式软件的开发工作一般是在主机上进行的。

那么当目标机的处理器与主机的处理器不同时（比如目标机是 ARM 处理器，而主机是 x86 处理器），如何保证在主机上编译的程序能在目标机上运行呢？

当编译 gcc 编译器时，需要指定主机处理器型号和目标机处理器型号，如果不加指定，则认为主机和目标机是一样的，且与当前编译它的环境一致。因此，在一台 Linux 主机上编译 gcc 如果不对主机和目标机处理器加以指定，则所编译出来的编译器将运行于同一处理器型号的 Linux 主机上，且采用这一编译器所编译出来的程序也将运行于同一处理器型号的 Linux 主机上。

如果在编译 gcc 时，指定主机处理器型号和目标机处理器型号不一致，则所生成的编译器就是交叉编译器（cross compiler），即所生成的编译器将在指定的主机上进行程序编译活动，但编译器生成的程序却运行于目标机上。

4.2　gcc 幕后工作揭示

让我们看看 gcc 是如何将图 4.1 中的 C 代码编译成可执行程序的。gcc 会对代码完成预处理（preprocessing）、编译（compilation）、汇编（assembly）和链接（linking）四个步骤。图 4.2 示例说明了这四个步骤，以及各步骤所使用到的 gcc 工具和产生的中间文件。

```
00001: #include <stdio.h>
00002:
00003: #define min(X, Y) ((X) < (Y) ? (X) : (Y))
00004:
00005: int main()
00006: {
00007:     printf ("The min is %d\n", min (3, 4));
00008:     return 0;
00009: }
```

图 4.1

首先，一个文件只要是以.c 结尾的，gcc 对之所做的第一件事就是调用 cpp 工具（C Preprocessor）对之进行预处理并生成一个新的文件（这里假设是 main.pre.c）。预处理的目的就是展开源文件中的所有宏指令。比如，对于所有以#include 指令包含的文件，被包含文件的内容将会放入新生成的文件中；所有引用#define 指令定义的宏在新生成的文件中将会被替换成宏的定义内容，等等。

接着，main.pre.c 文件被 gcc 调用 cc 工具（C and C++ compiler）对其进行编译，编译的结果是生成汇编程序文件（这里假设是 main.s）。编译，通俗地理解就是完成一定的翻译工作，将一种格式转换成另一种格式。当 C 程序被编译成汇编程序时，C 语言中的语法将被转换成汇

编语言中的相应元素，且在汇编语言中找不到任何 C 语言语法方面的踪迹。

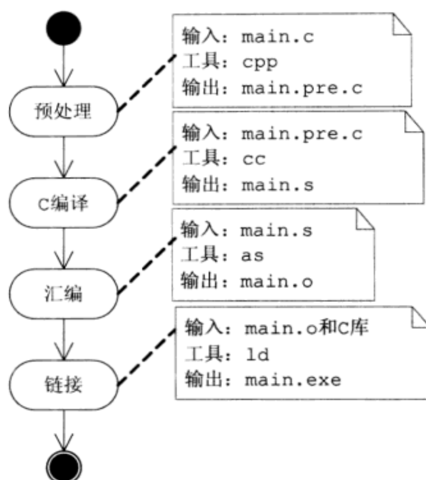


图 4.2

然后，gcc 使用 as（assembler）汇编器将 main.s 转换成目标文件（这里假设是 main.o）。汇编后获得的目标文件中只包含符号信息，而没有任何 C 语言和汇编语言语法方面的踪迹。

最后，gcc 使用 ld 链接器将 main.o 与 C 语言的标准库链接在一起，生成一个可以运行的可执行程序（这里假设是 main.exe）。在 Linux 世界里，可执行文件并不像 Windows 操作系统上的那样需要一个.exe 后缀，而是可以生成任何形式的文件名。本书对于可执行文件都采用.exe 文件结尾是便于读者理解。

由此看来，一个 C 源文件的编译过程需要经历两次编译，一次是将 C 语言转换成汇编语言，另一次是将汇编语言转换成目标文件。为此，在说到“编译”时我们需要根据情景理解具体是指哪一步（或同时包含两步）。

我们可以通过一定的方法来验证一个 C 源文件到底是被一步到位地编译成目标文件还是分成这里所说的两步。图 4.3 显示了通过对 as 工具进行更名从而使得 gcc 找不到它这种方式加以验证。注意，我们只使用-c 选项要求 gcc 只生成 main.o 而不进行链接。结果清楚地表明 gcc 是分两步完成的。

```
$ which as
/usr/bin/as

$ mv /usr/bin/as /usr/bin/as.bak

$ gcc -c main.c
gcc: error trying to exec 'as': execvp: No such file or directory
```

图 4.3

gcc 在整个编译过程中所需使用到的 as 和 ld 工具都来自于 binutils 工具集。本书之所以没有对 cpp 和 cc 两个工具进行介绍，是因为我们并不需要直接用到它们，而是使用 gcc 这个“前端”工具就行了。至于 as 和 ld 两个工具，后面有专门的章节加以介绍。

使用 gcc 进行程序编译其实隐含了这里介绍的四个步骤。后面为了表述方便，我们并不具体到 gcc 背后所使用的工具，这一点请读者在阅读后面的内容时注意。比如，如果说通过 gcc 进行链接，其本意是指通过 ld 工具进行链接。

4.3　实用的 gcc 选项

命令选项给人感觉有点微不足道，孰不知在软件开发过程中如果掌握一些选项将显著地提高解决问题的效率和帮助我们探寻更深层的知识。下面将要介绍的 gcc 选项，作者认为在工作中具有很强的实用性。

4.3.1　解决宏错误的好帮手

几乎每个源程序都得用到 C 语言中的宏，在软件开发过程中不可避免地会遇到与宏相关的编译错误。我们将图 4.1 的程序做一细微改动，将“min(X, Y)”宏定义写成“min (X, Y)”，即 min 后增加一个空格。然后进行编译，将获得图 4.4 所示的错误结果。

```
$ gcc main.c -o main.exe
main.c: In function 'main':
main.c:7: error: 'X' undeclared (first use in this function)
main.c:7: error: (Each undeclared identifier is reported only once
main.c:7: error: for each function it appears in.)
main.c:7: error: 'Y' undeclared (first use in this function)
```

图 4.4

读者或许对 gcc 所报告的错误能一下子就知道问题的根源，但是，有时即使是很简单的宏错误也不容易察觉。通过使用 gcc 的-E 选项能有效地帮助我们解决与宏相关的编译错误。

通过使用-E 选项，可以获得 gcc 对.c 文件完成预处理后的结果。图 4.5 示例说明了如何使用-E 选项获得 main.pre.c 文件。图 4.6 列出了 main.pre.c 文件的头尾两个部分。

```
$ gcc -E main.c > main.pre.c
```

图 4.5

```
00001: # 1 "main.c"
00002: # 1 "<built-in>"
00003: # 1 "<command-line>"
00004: # 1 "main.c"
00005: # 1 "/usr/include/stdio.h" 1 3 4
00006: # 29 "/usr/include/stdio.h" 3 4
```

```
00007: # 1 "/usr/include/_ansi.h" 1 3 4
00008: # 15 "/usr/include/_ansi.h" 3 4
00009: # 1 "/usr/include/newlib.h" 1 3 4
00010: # 16 "/usr/include/_ansi.h" 2 3 4
00011: # 1 "/usr/include/sys/config.h" 1 3 4
00012: ......
01139: # 683 "/usr/include/stdio.h" 3 4
01140:
01141: # 2 "main.c" 2
01142:
01143:
01144:
01145: int main()
01146: {
01147:     printf ("The min is %d\n", (X, Y) ((X) < (Y) ? (X) : (Y)) (3, 4));
01148:     return 0;
01149: }
```

图 4.6

从预处理结果的 1147 行可以看出，min 宏的问题一目了然。

从 main.pre.c 文件中我们可以看到存在大量的以“#”开头的行。每行的格式是“# 行号 文件名 标志”，其中的“行号”与“文件名”表示从它后一行开始的内容来源于哪一个文件的哪一行；标志可以是 1、2、3 和 4 四个数字，每个数字的含义如图 4.7 所示。

数字	含　义
1	表示一个新文件的开始
2	表示从一个被包含的文件中返回
3	表示后面的内容来自于系统头文件
4	表示后面的内容应当被当做一个隐式的`extern "C"'块

图 4.7

以图 4.6 中 main.pre.c 文件的 1141 行为例。它表示后面的内容来自于 main.c 文件的第 2 行（开始），且从被包含的“/usr/include/stdio.h”文件返回。对于“/usr/include/stdio.h”文件的包含是图 4.1 中的第 1 行指令所造成的。main.pre.c 文件的第 5 行指示了进入“/usr/include/stdio.h”文件，且从第 6 行到 1140 行的内容都来源于该文件。

这里所列的 main.c 文件只需要用到系统头文件，当它需要使用到非系统头文件且它们不在 main.c 所在目录内时，需要通过使用 gcc 的-I 参数加以指定，否则 gcc 会因为无法获得必要的头文件进行宏展开而报错。

4.3.2 辅助编写汇编程序的好方法

全面深入的嵌入式软件开发一定离不开编写汇编程序。如果真是要从无到有地编写汇编程序那还真不是一件容易事，好在通过 gcc 可以使得这一工作变得更简单。

前面在探讨 C 程序的编译步骤时指出，C 程序会被 cc 工具先编译成汇编程序。如果在编

写汇编程序时先使用 C 语言编写实现相应功能的函数，然后通过使用 gcc 获得函数所对应的汇编代码，再在此基础上做一定的修改，那将极大地简化汇编程序的编写工作。通过使用 gcc 的-S 参数能获得一个 C 源程序文件的汇编程序。对于图 4.8 所示的一个简单的 C 源程序，图 4.9 示例说明了如何通过-S 参数来获得与之对应的汇编程序。

```
foo.c
#include <stdio.h>

void foo ()
{
printf ("This is foo ().\n");
}
```

图 4.8

```
$ gcc -S -O2 foo.c

$ cat foo.s
        .file   "foo.c"
        .section .rdata,"dr"
LC0:
        .ascii "This is foo ().\0"
        .text
        .p2align 4,,15
.globl  foo
        .def    foo;    .scl    2;      .type   32;     .endef
 foo:
        pushl   ebp
        movl    esp,  ebp
        subl   $8,  esp
        movl   $LC0, (esp)
        call    puts
        leave
        ret
        .def    puts;  .scl    2;      .type   32;     .endef
```

图 4.9

使用-S 参数时，我们可以根据需要使用-O 优化选项。从 foo.s 的内容可以看出，“This is foo ().\0” 这个字符串是放在.rdata 段的。看来获得 C 程序对应的汇编代码还有助于我们了解 C 语言实现方面的细节。

4.3.3 获取系统头文件路径

在某些情形下，我们需要知道 gcc 所使用的系统头文件路径。系统头文件是指 C 语言本身和操作系统相关的头文件。后面第 30 章所讨论的静态分析工具就需要告之系统头文件路径。通过使用 gcc 的-v 选项就可以获取系统头文件路径，如图 4.10 所示。其中列出的 “search list” 就是指系统头文件路径。

```
$ gcc -v main.c
Using built-in specs.
```

```
Target: i686-pc-cygwin
……显示内容有删减……
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /usr/lib/gcc/i686-pc-cygwin/4.3.4/include
 /usr/lib/gcc/i686-pc-cygwin/4.3.4/include-fixed
 /usr/include
 /usr/lib/gcc/i686-pc-cygwin/4.3.4/../../../../include/w32api
End of search list.
……显示内容有删减……
```

图 4.10

在使用-v 选项获取系统头文件的路径时，必须后面带一个有效的 C 程序源文件，否则 gcc 输出的信息将不包含系统头文件路径这一内容，读者可以试试看。

4.3.4 产生映射文件

除了使用 5.3 节介绍的 nm 工具了解一个程序文件中各符号在内存中的布局信息外，还可以让 gcc 在生成可执行程序时为我们生成更为详细的映射文件。

实际上，映射文件的生成是通过 ld 链接器来做到的，我们可以通过使用 gcc 的选项来告诉链接器为我们生成映射文件，如图 4.11 所示。

```
$ gcc -Wl,-Map=main.map main.c -o main.exe

$ ls
main.c*  main.exe*  main.map
```

图 4.11

gcc 的-Wl 选项用于指定传递给链接器的选项，-Map=main.map 选项由 gcc 传递给链接器以指示链接器为我们生成名为 main.map 的映射文件。在使用-Wl 选项时，它后面的选项如有多个，则需用逗号加以分割。

映射文件中除了包含 nm 工具所获取的信息外，还包含各符号来源于哪一个库及库中的哪一个目标文件等更为详细的信息。读者可以打开 main.map 文件以查看详情。

4.3.5 通过选项定义宏

除了使用#define 指令在源文件中定义宏外，还可以在编译一个源文件时通过使用-D 选项定义宏。对于图 4.12 所示的源程序，图 4.13 示例说明了如何通过-D 选项来定义 GREETING 宏及程序的运行结果。

```
00001: #include <stdio.h>
00002:
00003: int main()
```

```
00004: {
00005:     printf ("The greeting is \"%s\".\n", GREETING);
00006:     return 0;
00007: }
```

图 4.12

```
$ gcc -D'GREETING="Hello"' main.c -o main.exe

$ ./main.exe
The greeting is "Hello".
```

图 4.13

4.3.6　生成依赖关系

第 3 章介绍的 make 需要通过依赖关系来决定每次构建时哪些文件需要重新编译。通过使用 gcc 的选项能让我们轻松地获得 make 所需的源文件依赖关系。

图 4.14 所示的源程序包含了三个文件：一个是来自标准库的 stdio.h，另外两个则是我们虚构的，分别是 main.h 和 foo.c。虚构的两个空文件通过 touch 命令创建，如图 4.15 所示。

```
00001: #include <stdio.h>
00002: #include "main.h"
00003: #include "foo.c"
00004:
00005: int main()
00006: {
00007:     printf ("Hello world!\n");
00008:     return 0;
00009: }
```

图 4.14

```
$ touch main.h foo.c
```

图 4.15

通过使用-M 选项，可以获得 main.o 文件的依赖关系，如图 4.16 所示。

```
$ gcc -M main.c
main.o: main.c /usr/include/stdio.h /usr/include/_ansi.h \
  /usr/include/newlib.h /usr/include/sys/config.h \
  /usr/include/machine/ieeefp.h /usr/include/sys/features.h \
  /usr/include/cygwin/config.h \
  /usr/lib/gcc/i686-pc-cygwin/4.3.4/include/stddef.h \
  /usr/lib/gcc/i686-pc-cygwin/4.3.4/include/stdarg.h \
  /usr/include/sys/reent.h /usr/include/_ansi.h \
  /usr/include/sys/_types.h /usr/include/machine/_types.h \
  /usr/include/machine/_default_types.h \
  /usr/include/sys/lock.h /usr/include/sys/types.h \
  /usr/include/machine/types.h /usr/include/cygwin/types.h \
  /usr/include/sys/sysmacros.h /usr/include/stdint.h \
```

```
/usr/include/endian.h /usr/include/byteswap.h \
/usr/include/sys/stdio.h \
/usr/include/sys/cdefs.h main.h foo.c
```

图 4.16

从图中的显示结果可以看到，gcc 输出了 main.c 所包含的所有文件，其中一部分是直接包含的，而另一部分是间接包含的。被包含的文件分成两类：一类来自系统头文件，比如_ansi.h、newlib.h 等；另一类来自我们所开发的项目，比如这里的 main.h 和 foo.c。

由于系统头文件在绝大多数情形下是不会改变的，因此在构造 make 所需的依赖关系时不必将它们纳入其中。通过使用-MM 选项，可以让 gcc 生成不包含系统头文件的依赖关系，如图 4.17 所示。

```
$ gcc -MM main.c
main.o: main.c main.h foo.c
```

图 4.17

4.3.7　指定链接库

当一个可执行程序的生成需要使用其他的库时，需要在链接时加以指定，这就要用到 gcc 的-L 和-l 选项。让我们以图 4.18 中的程序为例来看一看如何使用这两个选项，这里假设 foo.c 文件将被用于生成 libfoo.a 库。

```
foo.c
#include <stdio.h>

void foo ()
{
    printf ("This is foo ().\n");
}
```

```
extern void foo ();

int main ()
{
    foo ();
    return 0;
}
```

图 4.18

图 4.19 示例说明了生成 main.exe 可执行程序所需的步骤。注意，-L 选项用于告诉 gcc 可以从哪个目录查找库文件，可以多次使用它以指定多个目录；-l 选项则用于告诉 gcc 在生成可执行程序时需要链接的库名，这一选项同样可以多次使用以指定多个库。使用-l 选项时要注意，后面所跟的名字并不包括“lib”前缀和“.a”后缀。比如在这里的例子中，-lfoo 就是代表指定 libfoo.a 库参与链接。

有趣的是，图 4.19 显示链接器报告说找不到 foo()函数的定义。产生这一错误似乎与库在命令行的指定位置有关，图 4.20 示例说明了调整选项顺序后的编译结果。

```
$ gcc -c foo.c

$ ar crs libfoo.a foo.o

$ gcc -o main.exe -L. -lfoo main.c
/tmp/cc1BtxzC.o:main.c:(.text+0x17): undefined reference to `foo'
collect2: ld returned 1 exit status
```

图 4.19

```
$ gcc -o main.exe -L. main.c -lfoo

$ ./main.exe
This is foo ().
```

图 4.20

作者并不清楚为什么 gcc 会有这样的行为，但根据自己的经验发现，在使用 gcc 进行程序链接时依赖关系需要从左向右指定，具体含义需要通过例子进行解释。如果存在图 4.21 所示的程序，且假设其中的 foo.c 将被编译成 libfoo.a 库，以及 bar.c 将被编译成 libbar.a 库。

```
foo.c
#include <stdio.h>

extern void bar ();

void foo ()
{
   printf ("This is foo ().\n");
   bar ();
}
bar.c
#include <stdio.h>

void bar ()
{
   printf ("This is bar ().\n");
}

extern void foo ();

int main ()
{
   foo ();
   return 0;
}
```

图 4.21

图 4.22 分别列出了函数调用间的依赖关系和文件间的依赖关系。文件的依赖关系也指明了在指定链接库时的顺序关系，也就是说，gcc 选项中必须采用“main.c -lfoo -lbar”这样的顺序，

否则就会出现 gcc 报告不是 foo()函数没有定义就是 bar()函数没有找到这样的错误。图 4.23 示例说明了编译的步骤和程序的运行结果。

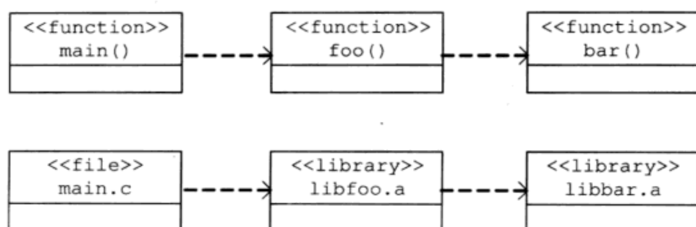


图 4.22

```
$ gcc -c foo.c

$ ar crs libfoo.a foo.o

$ gcc -c bar.c

$ ar crs libbar.a bar.o

$ gcc -o main.exe -L. main.c -lfoo -lbar

$ ./main.exe
This is foo ().
This is bar ().
```

图 4.23

gcc 的这种奇怪特性造成当依赖关系比较复杂时需要对同一个库在不同的位置指定多次，否则就会出现无法成功链接的情形。



第 5 章

binutils工具集，软件开发利器

如果使用 gcc 作为编译器，那么在开发过程中一定离不开使用与之配套的一个工具集[①]（tool chain），即 binutils。工具集中的部分工具除了被 gcc 在后台使用为我们创建程序文件[②]外，其他的则有助于方便开发和调试。

在 binutils 工具集中，以下工具是我们在做嵌入式软件开发时需要掌握的。

- as 是汇编编译器，用于将汇编代码转换为目标文件。在第 8 章已对其进行了介绍。
- addr2line 用于得到程序指令地址所对应的函数，以及函数所在的源文件名和行号。
- ar 用于创建和修改档案文件，以及从档案文件中抽取文件。静态库(.a 文件)就是一种档案文件，需要用它生成和管理。
- ld 是链接器，这是第 6 章的主题。
- nm 用于列出程序文件中的符号及符号在内存中的(开始)地址。符号包含 C 程序中的函数名和变量名。
- objcopy 可以用来从程序文件中拷贝出我们所指定的段，对于程序文件中的段请参见 9.1 节。在将引导加载器烧制到闪存中时，有时需要通过从程序中抽取段的方式生成烧写文件，这时 objcopy 工具就能派上用场。
- objdump 能显示程序文件的相关信息和对程序文件进行反汇编。
- ranlib 用于生成一个档案文件的内容索引，以加快对档案文件的查找速度。将该工具运用于静态库能提高库参与链接的效率。
- size 用于了解程序文件中各段的大小。
- strings 用于查看程序文件内的可显示字符串。
- strip 用于剥去程序文件的调试信息，以减小程序文件所占用的存储空间。这个工具对于存储空间有限的嵌入式系统尤为有用。

在不少嵌入式开发环境中，编译器的名称往往不是 gcc，而是像 arm-rtems-gcc 这样的名称。对于这种命名形式的编译器，读者通常可以找到 arm-rtems-addr2line、arm-rtems-objdump 等相应名称的工具，这是 GNU 工具集的一种命名惯例。

---

① 如果从英文原意翻译过来应是“工具链”，但作者认为中文的“集”更贴切。

② 本书所说的程序文件包含目标文件、库文件和可执行文件。

下面，让我们通过例子看一看各个工具的用处。需要注意的是，本文并不是 binutils 工具集的完整参考手册，对于每一个工具的讲解都是基于其实用性展开的。当读者需要得到更为详细的帮助信息时，可以参照相应工具的 man 和 info 信息[3]。另一种更为简单的方法是，运行相应的工具并指定--help 参数，可以获得该工具的简单帮助信息。

5.1　addr2line，指令地址翻译器

为了说明 addr2line 的作用，我们从图 5.1 所示的简单示例程序开始。

```
main.c
#include <stdio.h>

void foo ()
{
   printf ("The address of foo () is %p.\n", foo);
}

int main ()
{
   foo ();
   return 0;
}
```

图 5.1

像图 5.2 那样将 main.c 编译成可执行文件。编译时一定要带上-g 选项，这是为了让编译器在可执行文件中放置调试信息，否则 addr2line 命令将不能发挥作用。具体调试信息中包含了哪些内容，在讲解 objdump 时（参见 5.4 节）还会谈及。运行 test.exe 程序可以获得 foo()函数的（开始）地址。

```
$ gcc -g main.c -o test

$ ./test.exe
The address of foo () is 0x401100.
```

图 5.2

通过 foo()函数的开始地址，可以检查 addr2line 是如何起作用的，如图 5.3 所示。从命令的运行结果来看，addr2line 工具可以帮助指出程序地址所对应的函数、函数所在的文件名和文件行号。

```
$ addr2line 0x401100 -f -e test.exe
foo
/home/Administrator/main.c:4
```

图 5.3

③ 在 Cygwin 环境或 Linux 操作系统上，通过运行“man 工具名”的方式将获得工具的简单（但完整的）使用手册。运行“info 工具名”则能获得官方更为详细的说明。

这里的地址是通过程序打印而获得的，在现实工作中往往是程序崩溃时通过某种方式获得的，在这种情形下使用 addr2line 就可以了解崩溃发生点。另外，通过 nm 工具（参见 5.3 节）可以得到图 5.4 所示的信息。

```
$ nm -n test.exe
...... 显示结果有删减 ......
00401100 T  foo
0040111C T  main
```

图 5.4

从 nm 输出的信息可以看出，foo()函数所对应的开始地址为 0x00401100，而 foo()函数是有大小的，其大小就是函数所包含的实现其功能的指令所占的字节数。foo()函数的大小是_main 的地址减_foo 的地址。如果传给 addr2line 的地址是从 0x0040100 到 0x0040111C 的任一地址，情况会是怎样呢？图 5.5 是测试结果。

```
$ addr2line 0x401110 -f -e test.exe
foo
/home/Administrator/main.c:4

$ addr2line 0x40111B -f -e test.exe
foo
/home/Administrator/main.c:4
```

图 5.5

测试结果表明，addr2line 可以通过函数的任一地址找到所属函数的相关信息。

如果是 C++程序，运用 addr2line 又有什么不同呢？现在假设存在图 5.6 所示的一段 C++程序。

```
#include <iostream>

using namespace std;

void foo ()
{
   cout << "The address of foo () is " << hex << int (foo) << endl;
}

int main ()
{
   foo ();
   return 0;
}
```

图 5.6

采用 g++编译这段代码并运行，将得到图 5.7 所示的结果。

```
$ g++ -g main.cpp -o cpptest

$ ./cpptest.exe
The address of foo () is 401150
```

图 5.7

再使用 addr2line 查看地址 0x401150 所对应的函数，结果如图 5.8 所示。结果中似乎出现了乱码，本应是_foo 即变成了_Z3foov。

```
$ addr2line 0x401150 -f -e cpptest.exe
 Z3foov
/home/Administrator/main.cpp:6
```

图 5.8

乱码正体现了 C++语言的一个特点。在 GNU 工具集中存在"mangling"这样一个称呼，而在 Windows 中称为"decorating"，都是指对 C++中的函数名进行名字分裂。名字分裂是因为在 C++源程序中允许多个函数是重名的（即重载）。

C++语言是在 C 语言之上发展起来的，从 C 语言的角度来看并不存在重载的概念，因此在 C++内重载的函数从 C 语言的角度来看其名称必须不同。为了做到这一点，C++编译器的处理方法是对于每一个函数，将根据其输入参数采用一定的编码方式，形成不同的 C 函数名，这一过程就是名字分裂过程。正如上面所看到的，_Z3foov 其实就是 C++程序中 foo()函数的名字分裂后的形式。

使用 addr2line 的--demangle 选项可以获得我们所书写的函数名。从图 5.9（在 Cygwin 中运行的结果）可以看出，增加了--demangle 选项后，名字变成了 Z3foov。看来在 Cygwin 上这个选项不能完全起作用。

```
$ addr2line 0x401150 --demangle=gnu-v3 -f -e cpptest.exe
Z3foov
/home/Administrator/main.cpp:6
```

图 5.9

图 5.10 列出了在一台 Linux 主机上使用--demangle 选项的效果。

```
                                  g++ -g main.cpp -o cpptest
                                  ./cpptest.exe
The address of foo () is 80486e4
                                  addr2line 0x80486e4 --demangle=gnu-v3 -f -e cpptest.exe
foo()
/home/iUbuntu/main.cpp:6
```

图 5.10

5.2　ar，静态库生成器

如果要将多个.o 文件生成一个库文件，则存在两种类型的库。一种是静态库，在 Linux 的世界里其后缀是.a；另一种是动态库，其后缀是.so。

当可执行程序需要与静态库进行链接时，所使用到的库中的函数和数据会被拷贝到最终的可执行程序中。比如，如果 libx.a 中存在一个 foo()函数且程序 A 和程序 B 都需要用到它，那么在链接以后，程序 A 和 B 的可执行程序中都会存在 foo()函数代码的拷贝。

与静态库不同的是，采用动态库则不会生成多个拷贝。仍然以前面的例子为例，如果 foo()函数被放入了动态库 libx.so 中，则程序 A 和 B 中并不存在一个独立的 foo()函数代码拷贝，而是共享整个系统中唯一的一份。程序加载器在后台为我们将所需的动态库自动加载到内存中且保证整个系统只有一份拷贝。

如果一个系统中存在多个需要同时运行的程序且这些程序之间存在共享库，那么采用动态库的形式将更节省内存。但是，对于嵌入式系统，大多情形下都是整个软件就是一个可执行程序且不支持动态加载的方式，即以静态库为主。

binutils 中的 ar 被用来管理静态库。先看一看一个静态库中到底有些什么。下面拿系统库文件/lib/libc.a 为例以探究竟。将 libc.a 复制到另一个目录中，并对其采用 ar 的 x 参数进行解压操作，结果如图 5.11 所示。

```
$ cp /lib/libc.a libc.a

$ ls
libc.a

$ ar x libc.a

$ ls
...... 显示结果有删减 ......
acltotext.o     fsetpos.o      initgroups.o    regcomp.o      tmpfile.o
chown.o         fstat.o        lchown.o        regerror.o     truncate.o
ftello.o        libc.a         regexec.o       ftruncate.o    libcmain.o
regfree.o       dll_entry.o    getegid.o       lseek.o        seekdir.o
```

图 5.11

相信读者对于解压出来的.o 文件名不会陌生。每一个.o 文件差不多都能找到其文件名所对应的 C 库函数。采用 GNU 工具集进行开发时，一个静态库其实就是将.o 文件打包而生成的档案文件。

为了示例说明如何使用 ar 生成静态库，需要一些源程序文件。现在假设有 foo.c 和 bar.c 两个 C 程序文件，它们分别实现了 foo()和 bar()两个函数，代码如图 5.12 所示。

```
foo.c
#include <stdio.h>

void foo ()
{
   printf ("This is foo ().\n");
}
```

```
bar.c
#include <stdio.h>

void bar ()
{
   printf ("This is bar ().\n");
}
```

图 5.12

如果希望将 foo()和 bar()函数放入 libmy.a 库中，先要将它们分别编译成.o 目标文件，然后用 ar 命令来生成 libmy.a，如图 5.13 所示。其中，ar 的 c 参数表示创建一个档案文件，而 r 参数指示将文件增加到所创建的库文件中，s 参数是为了生成库索引以提高库被链接时的效率。

```
$ gcc -c foo.c

$ gcc -c bar.c

$ ar crs libmy.a foo.o bar.o
```

图 5.13

库一旦生成，我们可以用图 5.14 所示的程序检查其可使用性。图 5.15 示例说明了如何将之与库编译生成可执行文件，其中还示例说明了程序的运行结果。

```
extern void foo ();
extern void bar ();

int main ()
{
   foo ();
   bar ();
   return 0;
}
```

图 5.14

```
$ gcc main.c libmy.a -o mylib

$ ./mylib.exe
This is foo ().
This is bar ().
```

图 5.15

采用 ar 的 t 参数可以查看一个静态库中有什么内容，图 5.16 的操作示例说明了这个参数的作用。

```
$ ar t libmy.a
foo.o
bar.o
```

图 5.16

使用 d 参数可以删除库中的目标文件，图 5.17 示例说明了其效果。

```
$ ar d libmy.a foo.o

$ ar t libmy.a
bar.o
```

图 5.17

5.3　nm，符号显示器

总体说来，nm 用于列出程序文件中的符号。看看图 5.6 中生成的可执行程序使用 nm 工具能看到什么符号，如图 5.18 所示。

```
$ nm -n test.exe
...... 显示结果有删减 ......
00401100 T  foo
0040111c T  main
004011b8 T  printf
00401378 T  calloc
00401388 T  realloc
00401398 T  free
004013a8 T  malloc
00402000 D   data start
00402000 D  register frame info ptr
00402004 D  deregister frame info ptr
0040200c D   data end
```

图 5.18

nm 所列出的每一行由三部分组成。第一列是指程序运行时符号在内存中的地址，它表示函数或变量的开始地址；第二列是指相应的符号存放在哪一个段（参见 9.1 节）；最后一列则是符号的名称。

nm 列出的第二列信息非常有用，其意义在于可以让我们了解在程序中所定义的一个符号是被放在程序的哪一个段的。图 5.19 所示的表列出了常见字母的含义。

A	表示符号所对应的值是绝对的且在以后的连接过程中也不会改变
B 或 b	表示符号位于未初始化的数据段(.bss 段)中
C	表示没有被初始化的公共符号
D 或 d	表示符号位于初始化的数据段(.data 段)中
N	表示符号是调试用的
p	表示符号位于一个栈回溯段中
R 或 r	表示符号位于只读数据段(.rdata 段)中
T 或 t	表示符号位于代码段(.text 段)中
U	表示符号没有被定义

图 5.19

为了更清楚地理解 nm 所列出的符号与所编写程序之间的关系，需要通过图 5.20 所示的程序。我们还可以通过它来观察生成目标文件和可执行文件情形下 nm 的输出结果有何不同。

main.c
```c
#include <time.h>

int global1;
int global2 = 3;

static int static_global1;
static int static_global2 = 3;

void foo ()
{
    static int internal1;
    static int internal2 = 3;
    time (0);
}

static void bar ()
{
}

int main ()
{
    int local1;
    int local2 = 3;

    foo ();
    return 0;
}
```

图 5.20

图 5.21 是通过 nm 观察目标文件的结果。注意，输出结果中存在地址为 0 的符号。此时列出的地址由于程序还没有完成链接，所以是指符号在对应段中的相对偏移位置。另外，还可以看出_time 符号在文件中没有定义，因为它的实现位于 C 标准库 libc.a 内。

```
$ gcc -c -g main.c

$ nm -n main.o
         U  time
00000000 T  foo
00000000 D  global2
00000000 b  internal1.2129
00000004 d  static_global2
00000005 t  bar
00000008 d  internal2.2130
0000000a T  main
00000010 C  global1
00000010 b  static_global1
```

图 5.21

从 nm 的输出信息可以得出以下结论：

- 不论静态变量是定义在函数内还是函数外，程序段的分配方式都是一样的。如果静态变量是初始化好的，则会被分配在.data 段中，否则被分配在.bss 段中。
- 非静态的全局变量所分配的段只与其是否被初始化有关。如果被初始化了则被分配在.data 段中，否则被分配在.bss 段中。
- 函数无论是静态还是非静态的，总是被分配在.text 段中。字母“t”的大小写表示了符号是否是静态函数，小写表示静态。
- 函数内的局部变量由于是分配在栈上的（参见 10.4.1 节），所以在 nm 中看不到它们的身影。

图 5.22 示例说明了程序链接后 nm 的输出结果。最大的变化是所有的符号都有了具体的地址，而 global1 变量的分配空间也从前面的 C 变成了 B，_time 符号从无定义变成了分配在.text 段中。

```
$ gcc -g main.c -o test

$ nm -n test.exe
...... 显示结果有删减 ......
00401100 T  foo
00401114 t  bar
00401119 T  main
004011b4 T  time
0040200c D  global2
00402010 d  static_global2
00402014 d  internal2.1860
00404030 b  internal1.1859
00404040 b  static_global1
00404060 B  global1
```

图 5.22

nm 也像 addr2line 那样可以使用--demangle 参数，还原被分裂的函数名。最后，在 5.6 节将涉及 nm 的-s 参数，该参数用于查看库文件中的索引信息。

5.4　objdump，信息查看器

在嵌入式软件开发中，有时需要知道所生成的程序文件中的段信息以分析问题，或者需要查看 C 语言所对应的汇编代码，此时 objdump 工具就可以帮大忙了。

图 5.23 示例说明了如何使用 objdump 的-h 选项来查看图 5.22 生成的 test.exe 程序文件中的段信息。

```
$ objdump -h test.exe

test.exe:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000458  00401000  00401000  00000400  2**4
```

```
                             CONTENTS, ALLOC, LOAD, READONLY, CODE, DATA
  1 .data                    00000018  00402000  00402000  00000a00  2**2
                             CONTENTS, ALLOC, LOAD, DATA
  2 .rdata                   00000044  00403000  00403000  00000c00  2**2
                             CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .bss                     00000090  00404000  00404000  00000000  2**3
                             ALLOC
  4 .idata                   000001a4  00405000  00405000  00000e00  2**2
                             CONTENTS, ALLOC, LOAD, DATA
  5 .debug_aranges           00000020  00406000  00406000  00001000  2**0
                             CONTENTS, READONLY, DEBUGGING
  6 .debug_pubnames          00000071  00407000  00407000  00001200  2**0
                             CONTENTS, READONLY, DEBUGGING
  7 .debug_info              00000427  00408000  00408000  00001400  2**0
                             CONTENTS, READONLY, DEBUGGING
  8 .debug_abbrev            000000ed  00409000  00409000  00001a00  2**0
                             CONTENTS, READONLY, DEBUGGING
  9 .debug_line              000000b3  0040a000  0040a000  00001c00  2**0
                             CONTENTS, READONLY, DEBUGGING
 10 .debug_frame             0000006c  0040b000  0040b000  00001e00  2**2
                             CONTENTS, READONLY, DEBUGGING
 11 .debug_loc               0000009f  0040c000  0040c000  00002000  2**0
                             CONTENTS, READONLY, DEBUGGING
```

图 5.23

从 objdump 的输出信息还可以看出每一个段的大小，以及当程序运行时各段在内存中的开始地址。段地址存在 VMA（Virtual Memory Address，虚拟内存地址）和 LMA（Load Memory Address，加载内存地址）之别。对于每一个可加载的（loadable）或是可以重新分配的（re-allocatable）段，都存在一个 VMA 和一个 LMA。显然 VMA 的有效性离不开处理器和操作系统都支持内存管理单元。简单说来，VMA 指示的是在内存管理单元使能的情形下，段在程序运行时的开始地址；而 LMA 是指程序被加载时段在内存中的存放首地址。在大多数嵌入式系统中，VMA 和 LMA 是一样的。

在 objdump 的输出信息中还存在“File off”信息，它指明每一个段在程序文件（这里是 test.exe）中的存储位置。对于引导加载器来说，当加载程序时，就是要通过“File off”信息，从文件中读出相应段的内容，然后将这一内容写到段所指定的 VMA（内存）处。“Algn”指示了每一个段的边界对齐字节数是多少。

另外，从输出结果中还可以看出每一个段的属性，比如 READONLY、ALLOC 等。也可以看到很多段是以“.debug_”开头的，这些段是调试时需要使用到的，其中存储了程序中每一个符号的调试信息。这些调试信息采用了一定的编码格式，最为常用的格式是 DWARF（Debugging With Attributed Record Formats），DWARF 规范可以从网站 http://www.dwarfstd.org 上找到。使用 objdump 也可以查看程序文件中的 DWARF 信息，这需要用到-W 参数，图 5.24 示例说明了一个片段。

```
$ objdump -W test.exe
...... 显示结果有删减 ......
```

```
<0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
   < c>   DW_AT_producer      : GNU C 4.3.2 20080827 (beta) 2
   <2a>   DW_AT_language      : 1       (ANSI C)
   <2b>   DW_AT_name          : main.c
   <32>   DW_AT_comp_dir      : /home/Administrator
   <46>   DW_AT_low_pc        : 0x401100
   <4a>   DW_AT_high_pc       : 0x401142
   <4e>   DW_AT_stmt_list     : 0x0
...... 显示结果有删减 ......
<1><103>: Abbrev Number: 4 (DW_TAG_subprogram)
   <104>   DW_AT_external      : 1
   <105>   DW_AT_name          : foo
   <109>   DW_AT_decl_file     : 1
   <10a>   DW_AT_decl_line     : 10
   <10b>   DW_AT_low_pc        : 0x401100
   <10f>   DW_AT_high_pc       : 0x401114
   <113>   DW_AT_frame_base    : 0x0     (location list)
   <117>   DW_AT_sibling       : <0x14a>
```

图 5.24

尽管我们不熟悉 DWARF 规范，但可以看出调试信息中记录了源程序所在的路径、函数在内存中的起始地址（DW_AT_low_pc 和 DW_AT_high_pc）。正是因为其中存在每个函数的起始信息，所以采用 addr2line 工具能反向找到指令地址所对应的函数名、文件和文件行号。总而言之，所有在调试时能查看的符号信息都采用 DWARF 格式放在调试段中，包括局部变量在栈帧（参见 10.4.1 节）中的位置。当通过调试器查看一个变量的值时，调试器首先通过 DWARF 格式中的信息找到变量在内存中的地址，然后将内存中的内容根据变量的类型显示给我们看。

采用-d 选项可以显示程序文件的汇编代码，图 5.25 是采用-d 选项所显示 test.exe 程序文件的一个片段。

```
$ objdump -d test.exe
...... 显示结果有删减 ......
00401100 <_foo>:
  401100:       55                      push   %ebp
  401101:       89 e5                   mov    %esp,%ebp
  401103:       83 ec 08                sub    $0x8,%esp
  401106:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  40110d:       e8 a2 00 00 00          call   4011b4 <_time>
  401112:       c9                      leave
  401113:       c3                      ret
```

图 5.25

从显示的汇编来看，foo()函数的起始和终止地址分别是 0x401100 和 0x401113，与前面所显示的 DWARF 调试信息内所记录的信息是一致的。反汇编信息中还可以看出每个指令的机器码（地址与汇编代码中间部分的数据）。

在使用-d 选项进行反汇编时，另一个有用的选项是-S，它的作用是告诉 objdump 在反汇编时同时显示汇编代码所对应的 C/C++源程序，图 5.26 列出了使用这个选项的结果。

```
$ objdump -S -d test.exe
...... 显示结果有删减 ......
00401100 < foo>:

static int static_global1;
static int static_global2 = 3;

void foo ()
{
  401100:       55                      push    ebp
  401101:       89 e5                   mov     esp, ebp
  401103:       83 ec 08                sub     $0x8, esp
static int internal1;
static int internal2 = 3;
time (0);
  401106:       c7 04 24 00 00 00 00    movl    $0x0,(esp)
  40110d:       e8 a2 00 00 00          call    4011b4 < time>
}
  401112:       c9                      leave
  401113:       c3                      ret
```

图 5.26

采用将汇编与 C/C++源代码相结合显示的方式，有助于了解高级语言从汇编语言的角度是如何实现的。为了做到 C/C++代码与汇编代码的完全对应，不能对被编译程序使用优化选项(-O2 等)，否则会因为程序被优化而使得无法一一对应。objdump 也可以运用--demangle 选项，以帮助提高 C++程序在反汇编时的可读性。

采用-f 选项可以显示程序文件的头信息，如图 5.27 所示。其中的“start address”指示了可执行程序被执行时的入口地址是什么。入口地址即程序运行时的第一条指令在内存中的位置。显然，入口地址一定位于.text 段中。

```
$ objdump -f test.exe

test.exe:        file format pei-i386
architecture:    i386, flags 0x0000013a:
EXEC_P, HAS_DEBUG, HAS_SYMS, HAS_LOCALS, D_PAGED
start address 0x00401000
```

图 5.27

objdump 另一个非常有用的选项是-s，将它与-j 参数配合使用，能查看某一个段中的具体内容。图 5.28 示例说明了如何查看 test.exe 中.data 段的内容。

```
$ objdump -s -j .data test.exe

test.exe:     file format pei-i386

Contents of section .data:
 402000 00000000 [illegible] 00000000 03000000    ................
 402010 03000000 03000000                      ........
```

图 5.28

5.5 objcopy，段剪辑器

objcopy 可以对程序文件中的段进行过滤。先来看一看采用 objcopy 如何生成一个只包含.text 段的目标文件，仍以图 5.22 所生成的 test.exe 文件为例。通过-j 参数可以指定哪一个段是所需要抽取（拷贝）的，如图 5.29 所示，图中还通过 objdump 验证了 onlytext.exe 中的段是否只有.text。

```
$ objcopy -j .text test.exe onlytext.exe

$ objdump -h onlytext.exe

onlytext.exe:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000458  00401000  00401000  00000200  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE, DATA


$ objdump -f onlytext.exe

onlytext.exe:     file format pei-i386
architecture: i386, flags 0x00000132:
EXEC_P, HAS_SYMS, HAS_LOCALS, D_PAGED
start address 0x00401000
```

图 5.29

如果要指定多个段需要拷贝，比如，希望最后产生的 onlytext.exe（或许名字不应当再叫 onlytext 了）包含.text、.data 或.bss 段，那么可以使用多个-j 参数的方法，如图 5.30 所示。

```
$ objcopy -j .text -j .data -j .bss test.exe onlytext.exe
```

图 5.30

与-j 参数相反的是，采用-R 参数可以删除一个段。图 5.31 示例说明了通过-R 参数基于 test.exe 生成一个不包含.text 段的 notext.exe 文件。

```
$ objcopy -R .text test.exe notext.exe

$ objdump -h notext.exe

notext.exe:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .data         00000018  00402000  00402000  00000400  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  1 .rdata        00000044  00403000  00403000  00000600  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .bss          00000090  00404000  00404000  00000000  2**2
```

```
                           ALLOC
  3 .idata                 000001a4    00405000    00405000    00000800    2**2
                           CONTENTS, ALLOC, LOAD, DATA
  4 .debug_aranges         00000020    00406000    00406000    00000a00    2**0
                           CONTENTS, READONLY, DEBUGGING
  5 .debug_pubnames        00000071    00407000    00407000    00000c00    2**0
                           CONTENTS, READONLY, DEBUGGING
  6 .debug_info            00000427    00408000    00408000    00000e00    2**0
                           CONTENTS, READONLY, DEBUGGING
  7 .debug_abbrev          000000ed    00409000    00409000    00001400    2**0
                           CONTENTS, READONLY, DEBUGGING
  8 .debug_line            000000b3    0040a000    0040a000    00001600    2**0
                           CONTENTS, READONLY, DEBUGGING
  9 .debug_frame           0000006c    0040b000    0040b000    00001800    2**2
                           CONTENTS, READONLY, DEBUGGING
 10 .debug_loc             0000009b    0040c000    0040c000    00001a00    2**0
                           CONTENTS, READONLY, DEBUGGING
```

图 5.31

为了减小程序文件所占用的存储空间，可以将程序文件中的调试信息去除，最为常用的方法是通过使用 strip 命令（参见 5.9 节）。另外，采用这里的 objcopy 配合--strip-debug 选项也可以达到同样的目的。下面看一看采用这一选项对 notext.exe 进行操作后的结果是什么，如图 5.32 所示。

```
$ objcopy --strip-debug notext.exe

$ objdump -h notext.exe

notext.exe:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .data         00000018  00402000  00402000  00000400  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  1 .rdata        00000044  00403000  00403000  00000600  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .bss          00000090  00404000  00404000  00000000  2**3
                  ALLOC
  3 .idata        000001a4  00405000  00405000  00000800  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

图 5.32

objcopy 最为重要的功能就是能按照需要抽取程序文件中的段。在有的嵌入式系统中，比如制作引导加载器时就需要用到 objcopy，以便将代码段抽取出来，然后将其“烧”到系统的启动运行地址处（通常是一块闪存）。objcopy 还提供其他的一些有用的功能，比如，改变段的地址，但在作者的工作经验中没有用过这些功能，所以在此也不多讲，读者在需要时可以参照 objcopy 的帮助信息。

5.6　ranlib，库索引生成器

ranlib 的功能相对简单，就是用于在档案文件中生成文件索引，如图 5.33 所示。前面在讲

ar 时提到的 s 参数也具有同样的功能。当档案文件增加了索引后，对其内文件的存取速度将更快。如果档案文件是一个静态库，那么生成索引后的库链接速度更快。

```
$ ranlib libmy.a
```

图 5.33

可以用 nm 加上-s 参数来查看档案文件中的索引信息，如图 5.34 所示。

```
$ nm -s libmy.a

Archive index:
 bar in bar.o
 foo in foo.o

bar.o:
00000000 b .bss
00000000 d .data
00000000 r .rdata
00000000 t .text
00000000 T  bar
         U  puts

foo.o:
00000000 b .bss
00000000 d .data
00000000 r .rdata
00000000 t .text
00000000 T  foo
         U  puts
```

图 5.34

5.7 size，段大小观察器

size 工具被用于查看程序文件中各段的大小，图 5.35 示例说明了其功用。

```
$ size test.exe
   text    data     bss     dec     hex filename
   1180     444     144    1768     6e8 test.exe
```

图 5.35

在 5.4 节讲解 objdump 时看到，objdump 所显示的段除了.text、.data 和.bss 三个段外，还有.rdata 和.idata 两个段。在 size 所显示的段信息中，.rdata 段被归到.text 段中，而.idata 段被归到.data 段中。如果采用-A 选项，size 将显示出更详细的段信息，如图 5.36 所示。

```
$ size -A test.exe
test.exe  :
section              size      addr
.text                1112      4198400
```

```
.data                24        4202496
.rdata               68        4206592
.bss                 144       4210688
.idata               420       4214784
.debug_aranges       32        4218880
.debug_pubnames      113       4222976
.debug_info          1063      4227072
.debug_abbrev        237       4231168
.debug_line          179       4235264
.debug_frame         108       4239360
.debug_loc           155       4243456
Total                3655
```

图 5.36

5.8 strings，字符串窥视器

strings 用于查看程序文件中的可显示字符。图 5.37 的示例程序将用于帮助我们观察 strings 的功能。请注意其中定义的版本和密码信息。

```
#include <stdio.h>

#define VERSION    "2.3.7"

#define PASSWORD    "admin"

const char *get_password ()
{
   return PASSWORD;
}

int main ()
{
   printf ("Version: %s\n", VERSION);
   printf ("Password: %s\n", get_password ());
   return 0;
}
```

图 5.37

编译示例程序并运行，输出如图 5.38 所示。

```
$ gcc -g string.c string.exe

$ ./string.exe
Version: 2.3.7
Password: admin
```

图 5.38

现在用 strings 工具看看 string.exe 中有些什么字符信息。结果如图 5.39 所示。

```
$ strings string.exe
$h0@
```

```
$h0@
$L0@
$Y0@
%|P@
C$0@@
C@`@@
$10@
cyggcc s-1.dll
  register frame info
  deregister frame info
admin
2.3.7
Version: %s
Password: %s
  main
 impure ptr
calloc
cygwin internal
dll crt0  FP1lper process
free
malloc
printf
realloc
GetModuleHandleA
GetProcAddress
cygwin1.dll
KERNEL32.dll
```

图 5.39

是否发现版本和密码信息都可以从 strings 的输出结果中找到？还有，输出结果中有很多的函数名。

显然，通过 strings 工具，可以窥视到程序文件中的很多信息。即使是将程序文件中的调试信息给去除了，这些信息也照样可以被看到，因为这些信息是放在.data 段（或.rdata 段）中的。在软件开发过程中，我们可能会使用__FILE__宏，这些宏就会使得相应文件名的字符串被放入到.rdata 段中。在程序中定义的字符串，同样也会出现在.data 或.rdata 段中。这些信息都会向他人泄露程序的“痕迹”。

如果想将他人设计的程序作为自己程序的一部分而又不想让他人知道，那得小心了，因为通过 strings 可能查看到这些信息。另外，如果程序中需要定义密码等加密信息，则最好不要用字符串的形式，或者定义了也不要直接用它作为密码，而应当采用一定的算法对字符串进行再加工，使他人即使是从 strings 的输出结果中看到了这一信息，也无法猜出它。

strings 对于调试也是很有用的。比如，发布了一个软件到现场，但并不知道其版本是多少。如果程序中存在版本信息的字符串，那就可以通过 strings 获得程序的版本信息。

我们可以想象 strings 工具与具体的处理器是无关的。也就是说，可以用在 x86 处理器上运行的 strings 程序去查看在 PowerPC 上运行的程序文件中的字符信息。

5.9　strip，程序文件瘦身器

strip 用于去除程序文件中的调试信息以便减小程序文件的大小。它的功能与 objcopy 带--strip-debug 参数时的功能是一样的，这一点在前面也有提及。strip 所具有的功能，objcopy 也都有。



第 6 章

ld，链接器

链接器（linker）的功能，是将一个可执行程序所需的目标文件和库最终整合为一体。在第 9 章介绍程序中的段时将谈到，一个程序通常包含传统的三个段：.text、.data 和.bss 段。实际上，在目标文件和库被整合成一个可执行程序文件之前，通常各目标文件和库中也包含这三个段。不难想象链接器的功能其实就是将各个目标文件和库中的三个段进行合并。图 6.1 示例说明了链接器将两个目标文件链接成一个可执行程序文件的效果。

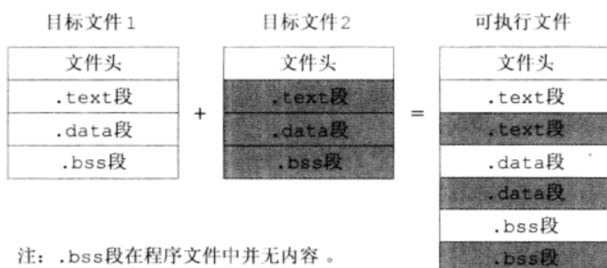


图 6.1

链接器所完成的链接工作并非只是简单地将各个目标文件或库中的段简单地堆砌在一起，而是还要完成一个被称为“重定位（relocation）”的工作。

6.1 重定位的概念

链接而生成的可执行程序虽然是放在文件中的，但当程序运行时需要加载到内存中。各段应放在内存空间的什么位置是由可执行程序文件内的头部信息指定的，至于这些信息从哪来本章的后面会给出答案。

一个程序一旦被加载到内存中，就意味着不论是函数还是变量，它们都会在内存中占据一定的内存空间，而这关系到内存地址。假设 foo()函数在加载到内存中后其地址刚好位于 0x10000 处。从处理器的角度来看，当我们在 C 程序中写下一行调用 foo()函数的语句时，意味着在调用 foo()函数时需要跳转到 0x10000 的内存地址处，那如何知道调用 foo()函数时应当跳转到 0x10000 地址处呢？这就是链接器所需要完成的工作。

当一个源文件被编译成目标文件时，此时目标文件中的各段并没有具体的地址，在目标文件中只记录了程序中的符号和各符号在段中的相对位置。当链接器将所有的目标文件整合成一个可执行程序文件时，各目标文件中的各段将会真正获得在内存中的具体地址。链接生成可执行程序时，需要根据每一个符号所对应的真实地址而更新相应的指令，从而实现真正的函数调用和变量引用功能，这一动作就是重定位。

链接器如何知道每一个目标文件中的各个段应当放在哪一地址处呢？这需要通过链接脚本来指定。

6.2 链接脚本

为了了解链接脚本中的内容到底是什么，可以使用 ld 的--verbose 选项导出 ld 的默认链接脚本。图 6.2 示例说明了如何导出 ld 的默认脚本，而图 6.3 则示例说明了所导出脚本的部分内容。

```
$ ld --verbose > ldscript
```

图 6.2

```
00003: /* Script for ld --enable-auto-import: Like the default script except
       read only data is placed into .data */
00004: SECTIONS
00005: {
00006:     /* Make the virtual address and file offset synced if the
00007:        alignment is lower than the target page size. */
00008:     . = SIZEOF_HEADERS;
00009:     . = ALIGN(__section_alignment__);
00010:     .text __image_base__ + ( __section_alignment__ < 0x1000
           ? . : __section_alignment__ ) :
00011:     {
00012:        *(.init)
00013:        *(.text)
00014:        *(SORT(.text$*))
00015:        *(.text.*)
00016:        *(.glue_7t)
00017:        *(.glue_7)
00018:        ___CTOR_LIST__ = .; __CTOR_LIST__ = . ;
00019:        LONG (-1);*(.ctors); *(.ctor); *(SORT(.ctors.*)); LONG (0);
00020:        ___DTOR_LIST__ = .; __DTOR_LIST__ = . ;
00021:        LONG (-1); *(.dtors); *(.dtor); *(SORT(.dtors.*)); LONG (0);
00022:        *(.fini)
00023:        /* ??? Why is .gcc_exc here? */
00024:        *(.gcc_exc)
00025:        PROVIDE (etext = .);
00026:        *(.gcc_except_table)
00027:     }
00028:     /* The Cygwin32 library uses a section to avoid copying certain data on
00029:        fork. This used to be named ".data". The linker used to include this
00030:        between __data_start__ and __data_end__, but that breaks building the
00031:        cygwin32 dll. Instead, we name the section ".data_cygwin_nocopy" and
00032:        explictly include it after __data_end__. */
00033:     .data BLOCK(__section_alignment__) :
```

```
00034:     {
00035:         __data_start__ = . ;
00036:         *(.data)
00037:         *(.data2)
00038:         *(SORT(.data$*))
00039:         *(.rdata)
00040:         *(SORT(.rdata$*))
00041:         *(.jcr)
00042:         __data_end__ = . ;
00043:         *(.data_cygwin_nocopy)
00044:     }
00045:     .rdata BLOCK(__section_alignment__) :
00046:     {
00047:         ___RUNTIME_PSEUDO_RELOC_LIST__ = .;
00048:         __RUNTIME_PSEUDO_RELOC_LIST__ = .;
00049:         *(.rdata_runtime_pseudo_reloc)
00050:         ___RUNTIME_PSEUDO_RELOC_LIST_END__ = .;
00051:         __RUNTIME_PSEUDO_RELOC_LIST_END__ = .;
00052:     }
00053:     .eh_frame BLOCK(__section_alignment__) :
00054:     {
00055:         *(.eh_frame)
00056:     }
00057:     .pdata BLOCK(__section_alignment__) :
00058:     {
00059:         *(.pdata)
00060:     }
00061:     .bss BLOCK(__section_alignment__) :
00062:     {
00063:         __bss_start__ = . ;
00064:         *(.bss)
00065:         *(COMMON)
00066:         __bss_end__ = . ;
00067:     }
00068:     ......
00120:     .endjunk BLOCK(__section_alignment__) :
00121:     {
00122:         /* end is deprecated, don't use it */
00123:         PROVIDE (end = .);
00124:         PROVIDE ( _end = .);
00125:         __end__ = .;
00126:     }
00127:     ......
00212: }
```

图 6.3

脚本中的内容看过去既熟悉又陌生。熟悉是因为其中包含了.text、.data、.bss 等我们所熟悉的段名，但也包含了很多我们不熟悉的段名（比如.init、.fini 等）和脚本命令（比如 SECTIONS、BLOCK 等）。

链接脚本的功能就是告诉链接器，如何将各个不同的目标文件（包括库）中的段合在一起并最终生成一个可执行程序（文件）。从链接脚本的角度来看，一方面它需要描述输出，即最终输出到可执行程序文件中的段；另一方面又要描述输入，即来自各个目标文件中的段。

下面让我们从一个简单的链接脚本开始学习链接脚本的语法。

6.2.1　段

一个可执行程序最终是分成多个段的。在嵌入式系统中，当引导加载器（参见第 19 章）加载一个可执行程序时，会根据文件中的头信息将各个段位于文件中的内容拷贝到内存中。至于拷贝到内存的哪一个具体地址是通过链接脚本指定的。

图 6.4 示例说明了一个很简单的链接脚本。注意，SECTIONS 命令表示定义可执行程序中的各个段。

```
00001: SECTIONS
00002: {
00003:     . = 0x1000000;
00004:     .text :
00005:     {
00006:         *(.text)
00007:     }
00008:     .data 0x8000000:
00009:     {
00010:         *(.data)
00011:     }
00012:     .bss :
00013:     {
00014:         *(.bss)
00015:     }
00016: }
```

图 6.4

在进一步介绍段之前，我们需要了解什么是位置指针。链接器的最终目的，是要将各个目标文件中的段合在一起，而合起来时各部分放在内存的什么位置就有了地址的概念。通过使用和操控位置指针就能实现对各个段在地址空间的安排，位置指针的值其实代表的就是处理器的地址空间。

SECTIONS 命令描述的是可执行程序各段在内存中的布局。在默认情形下，一进入 SECTIONS 命令的区间（第 2 行和第 16 行两个花括号之间的区域），位置指针的值就为 0。

第 3 行是一个改变位置指针的语句。其中的“.”就表示位置指针，通过将位置指针设置成 0x1000000 之后，后面的.text 段将从内存地址 0x1000000 处开始存放。从这条语句来看，位置指针与 C 语言中的指针概念是相似的，我们既可以改变位置指针的值，也可以获取它的值。

第 4～7 行是第一个输出段描述语句块，它表示生成的可执行程序中将有一个.text 段。由于.text 段所处的位置指针值为 0x1000000，因此当程序被加载时位于可执行程序中.text 段的内容将被拷贝到 0x1000000 内存地址开始处。输出到可执行程序的段中又包含什么呢？这是通过第 6 行的输入段描述来指定的。这一行的意思是，所有目标文件中的.text 段都将放入可执行程序的.text 段中。在图 6.3 中，第 12～26 行都用于指定可执行程序的.text 段中包含目标文件的哪些段。

第 8～11 行定义了可执行程序中将有一个.data 段。注意，这里采用了另外一种方法来改变位置指针。在第 8 行的.data 后面通过指定值 0x8000000 的方式，指明.data 段在程序运行时所存放的开始地址是 0x8000000。第 10 行的输入段描述指定所有库和目标文件中的.data 段。

第 12～15 行定义了可执行程序中的.bss 段。在这个链接脚本中，由于.bss 段并没有指定其地址值，因此它的位置完全取决于程序在链接时位置指针的值，而位置指针的值取决于它之前.data 输出段的大小。也就是说，.bss 段在内存中的开始位置是由 0x8000000 加上.data 段的大小而得出的。

当链接器生成可执行程序时，会将每一个段在内存中的开始地址及段大小这些信息放在可执行程序文件的头部，而这些信息是为程序加载器加载程序所准备的。

6.2.2　符号

在有些情形下，我们需要了解各个段在内存中的具体位置。比如，可能需要知道.bss 段在内存中的位置，以便对其进行置 0 初始化。在嵌入式系统中，当程序被引导加载器加载后，也需要一种方法指明哪些内存可以被当做堆（参见 9.2 节）。链接脚本中除了指定段和段在内存中的地址外，还可以定义符号，图 6.5 示例说明了如何定义符号。

```
00001: SECTIONS
00002: {
00003:     . = 0x10000;
00004:     .text :
00005:     {
00006:         *(.text)
00007:     }
00008:     .data :
00009:     {
00010:         *(.data)
00011:     }
00012:     .bss :
00013:     {
00014:         __bss_start__ = .;
00015:         *(.bss)
00016:         __bss_end__ = .;
00017:     }
00018:     __end__ = .;
00019: }
```

图 6.5

在第 14、16 和 18 行分别定义了三个符号。__bss_start__符号代表了.bss 段的开始地址，__bss_end__符号代表了.bss 段的结束地址。准确地说，__bss_end__符号所代表的地址并不属于.bss 段（还得减 1）。而__end__符号代表了程序各段所占用内存空间的结束地址，或者说它代表了堆的开始地址。这些符号的值是通过使用位置指针加以指定的。符号一旦在链接脚本中定义后，就可以在程序中使用它们，我们可以通过图 6.6 中的小程序加以试验。

```
main.c
00001: #include <stdio.h>
00002:
00003: extern char _bss_start__[];
00004: extern char _bss_end__[];
00005: extern char _end__[];
00006:
00007: int main ()
00008: {
00009:     printf ("__bss_start__ = 0x%p\n", _bss_start__);
00010:     printf ("__bss_end__ = 0x%p\n", _bss_end__);
00011:     printf ("__end__ = 0x%p\n", _end__);
00012:     return 0;
00013: }
```

图 6.6

在这个示例程序中，读者需要注意所声明的变量与链接脚本中定义的符号相差一个下划线前缀。这是因为在作者的环境中编译器默认会在 C 语言中的符号之前加上一个下划线。当然，可能有些操作系统上的 gcc 没有这一特性，如果是这样，则需要对声明的变量进行名字调整。请注意，尽管在此三个变量被声明成外部的字符数组，但是它还可以被声明成其他类型。

从图 6.3 可以看出，ld 的默认脚本中也定义了__bss_start__、__bss_end__和__end__三个符号，因此我们可以直接使用 gcc 对图 6.6 中的程序进行编译，而不需要编写额外的链接脚本。图 6.7 示例说明了如何编译测试程序，并显示了程序的运行结果和映射文件中的内容。

```
$ gcc -Wl,-Map=main.map main.c -o main.exe

$ ./main.exe
__bss_start__ = 0x0x404000
__bss_end__ = 0x0x404130
__end__ = 0x0x406000

$ vi main.map
……显示结果有删减……
.bss            0x00404000      0x130
                0x00404000                __bss_start__ = .
 *(.bss)
                0x00404130                __bss_end__ = .
……显示结果有删减……
.endjunk        0x00406000      0x0
                0x00406000                PROVIDE (end, .)
                0x00406000                PROVIDE (_end, .)
                0x00406000                __end__ = .
……显示结果有删减……
```

图 6.7

6.2.3 存储区域

在默认情形下，ld 认为整个可执行程序都是放入同一个存储空间的。如果一个嵌入式系统中存在多块不同的存储空间，就得使用到 MEMORY 命令进行存储区域定义。图 6.8 示例说明

了如何使用 MEMORY 命令，以及如何将程序的不同段放入不同的存储区域中。

```
00001: MEMORY
00002: {
00003:     RAM0 (WX) : ORIGIN = 0x40000000, LENGTH = 256K
00004:     RAM1 (WX) : ORIGIN = 0, LENGTH = 2M
00005: }
00006:
00007: SECTIONS
00008: {
00009:     .text :
00010:     {
00011:         . += 0x10000;
00012:         *(.text)
00013:     } > RAM1
00014:     .data :
00015:     {
00016:         *(.data)
00017:     } > RAM1
00018:     .bss :
00019:     {
00020:         *(.bss)
00021:     } > RAM0
00022: }
```

图 6.8

RAM0 和 RAM1 都被定义成可以读写和执行的存储区域。在 SECTIONS 命令中，将.bss 段放入了 RAM0 存储区域中，而将.text 和.data 段放入了 RAM1 存储区域中。

使用存储区域时，如果链接器碰到存放段大于存储区域的容量时就会发出告警。我们可以利用链接器的这一特性，通过定义多个（连续的）存储区域的形式监视各段是否超出规定的大小。

6.2.4　常用命令

在掌握了链接脚本中的段、符号和存储区域这三个概念后，就可以读懂很多的链接脚本了。除了这三个概念外，在链接脚本中还可以使用其他的一些命令以便更有效地编写。下面将介绍几个在嵌入式开发中常用到的命令。

如果读者在工作中碰到了这里没有介绍的命令，请查看 ld 的官方使用手册《The GNU Linker》。该手册从附书光盘中可以找到，其文件名为 ld.pdf。

6.2.4.1　ALIGN 和 BLOCK 命令

ALIGN 命令的格式是：

```
ALIGN (_align)
ALIGN (_exp, _align)
```

第一个 ALIGN 命令将返回位置指针之后的第一个满足边界对齐字节数_align 的地址值。

注意，ALIGN 并不改变位置指针的值。

第二个 ALIGN 命令返回_exp 表达式值之后的、满足边界对齐字节数_align 的地址值。显然，ALIGN(_align)等价于 ALIGN(., _align)。

BLOCK 命令与只有一个参数的 ALIGN 命令的作用是一样的，它的存在是为了兼容老的语法。

6.2.4.2 BYTE、SHORT、LONG 和 QUAD 命令

这些命令的格式是：

```
BYTE (_value)
SHORT (_value)
LONG (_value)
QUAD (_value)
```

这四个命令依次表示在输出的可执行程序文件中放置所占存储空间为 1、2、4 和 8 字节的值，值由_value 参数指定。图 6.9 所示的链接脚本将使得.text 段与.data 段之间存在 4 字节的“空隙”，且它的值为 1。

```
SECTIONS
{
    .text :
    {
        *(.text)
    }
    LONG(1)
    .data :
    {
        *(.data)
    }
}
```

图 6.9

在图 6.3 中的第 19 和 21 行分别使用 LONG 命令构建 C++中构造函数和析构函数的函数指针数组，各数组的第一项为-1，而最后一项为 0。

6.2.4.3 ENTRY 命令

ENTRY 命令的格式是：

```
ENTRY (_symbol)
```

这个命令用于指定可执行程序的入口点。入口点是指程序被加载到内存后，运行第一条指令所在的内存地址。

6.2.4.4 MEMORY 命令

MEMORY 命令的格式是（注：方括号中的内容表示是可选的）：

```
MEMORY
{
    name [(_attr)] : ORIGIN = _origin, LENGTH = _len
    ...
}
```

或

```
MEMORY
{
    name [(_attr)] : org = _origin, l = _len
    ...
}
```

图 6.8 示例说明了如何使用 MEMORY 命令。其中的_attr 可取值如图 6.10 所示。注意，它是大小写不敏感的。

_attr	含　义
R	只读段
W	可读写段
X	可执行段
A	可分配段
I	已初始化段
L	与 I 一样
!	反转以上任一选项的含义

图 6.10

当各段没有像图 6.8 那样通过使用“>”指定它在存储区域的位置时，ld 会自动根据各段的属性放入不同的存储区域中。对于图 6.11 所示的存储区域定义，ld 会自动将.text 段放入 rom 中，因为 rom 具有只读和可执行属性；另外，会将.data 和.bss 段放入 ram 中，因为它具有除可执行外的其他所有属性。

```
MEMORY
{
   rom (rx) : ORIGIN = 0, LENGTH = 256K
   ram (!x) : org = 0x40000000, l = 4M
}
```

图 6.11

MEMORY 命令中的 ORIGIN 是指存储区域的开始地址，而 LENGTH 是指存储区域的字节大小。

6.2.4.5　PROVIDE 命令

PROVIDE 命令的格式是：

```
PROVIDE (_symbol = _expression)
```

在某些情形下，我们希望脚本中定义的符号只有当它被引用时才出现。这可以通过PROVIDE命令实现。有些gcc对于C语言中所出现的符号在符号表中会在它的前面加上一个下划线，但有的又不会。对于图6.6所示的引用_bss_start__等符号的示例程序，可以通过使用PROVIDE命令保证该示例程序无论gcc的行为如何都能被成功编译，如图6.12所示。

```
00001: SECTIONS
00002: {
00003:     . = 0x10000;
00004:     .text :
00005:     {
00006:         *(.text)
00007:     }
00008:     .data :
00009:     {
00010:         *(.data)
00011:     }
00012:     .bss :
00013:     {
00014:         PROVIDE (__bss_start__ = .);
00015:         PROVIDE (_bss_start__ = .);
00016:         *(.bss)
00017:         PROVIDE (__bss_end__ = .);
00018:         PROVIDE (_bss_end__ = .);
00019:     }
00020:     PROVIDE (__end__ = .);
00021:     PROVIDE (_end__ = .);
00022: }
```

图6.12

6.2.4.6　SECTIONS命令

SECTIONS命令用于告诉ld如何将各个目标文件中的段映射到输出的可执行程序文件中，以及如何在内存中布局可执行程序中的各个段。该命令的格式是：

```
SECTIONS
{
    sections-command
    sections-command
    ...
}
```

命令体中的sections-command可以是：

（1）ENTRY命令。注意，ENTRY命令既可以放在SECTIONS命令体内，也可以放在体外。

（2）符号定义和赋值。

（3）输出段描述（output section description）。

如果在链接脚本中不使用 SECTIONS 命令，则 ld 将在可执行程序中创建与所有目标文件中同名的段并将同名的段合在一起。各段在可执行程序中的顺序与 ld 首次遇到它的顺序是一致的。另外，第一个段的地址会是 0。

输出段描述的格式如下：

```
section [_address] [(_type)] :
[AT (_lma)]
[ALIGN (_section_align)]
[SUBALIGN (_subsection_align)]
[_constraint]
{
   output-section-command
   output-section-command
   ...
} [> _region] [AT> _lma_region] [:_phdr :_phdr ...] [= _fillexp]
```

_section 指的是像.text、.data、.bss 这样的出现在可执行程序中的段名。"/DISCARD/"（不包含引号）是一个特殊的段名，它表示段内 output-section-command 所描述的各目标文件中的段需要被丢弃，而不输出到可执行程序文件中。

_address 指的是程序运行时段在内存中的虚拟地址。在大多数嵌入式系统中，只需指定这一地址就行了。可以结合使用 ALIGN 等命令，使得段的开始地址满足 MMU 的页边界对齐等要求。

_type 可以是图 6.13 所示表中的任一值。

_type	含 义
NOLOAD	当程序被运行时，段不被加载到内存中
DESCT、COPY、INFO、OVERLAY	这些值很少被使用，它们是为了兼容老的语法格式。它们都用于表示程序加载时段所需的内存是不能通过分配的形式获得的

图 6.13

_lma 指的是程序加载时段的加载地址，如果不指定则认为_lma 与_address 是一样的。大多的嵌入式系统都不指定它。_section_align 指示段在内存中存放的开始位置所应满足的边界对齐字节数。_subsection_align 则表示段中的子段所应满足的边界对齐字节数。_constraint 的值可以是 ONLY_IF_RO 和 ONLY_IF_RW，这两个值分别表示所有的输入段是只读和所有的输入段是可读写时才创建输出段。

输出段描述中的 output-section-command 可以是：

（1）符号定义和赋值。

（2）输入段描述（input section description）。

（3）值定义（参见 6.2.4.2 节）。

_region 和_lma_region 分别指使用 MEMORY 命令定义的存储区域的名称。一个是针对 VMA 的，而另一个是针对 LMA 的，它们都用于指明可执行程序运行时段应放入的存储区域。对于可执行程序中的每一个段，ld 都会在程序文件中为之创建一个段头，以便程序加载器在加载该段时使用。_phdr 的值将被用于设置段头。_fillexp 指示当段与下一个段之间存在内存“空隙”时，对“空隙”中的内存空间所需填入的值。图 6.14 示例说明了如何使用它将空隙中的各字节填充为 0x90。段与段之间的空隙通常是因为各段的开始地址需要满足一定的边界对齐要求而造成的。

```
SECTIONS
{
   .text :
   {
      *(.text)
   } = 0x90909090
}
```

图 6.14

我们使用输出段描述来告诉 ld 如何将程序布置到内存中，而输入段描述用来告诉 ld 如何将所有目标文件中的段映射到内存中。不难发现，链接脚本中的内容大部分是输入段描述。图 6.15 所示是输入段描述的常见格式。

```
格式1:  *(.text)
格式2: data.o(.data)
格式3: *(EXCLUDE_FILE (*crtend.o *otherfile.o) .ctors)
格式4: libcore.a:task.o(.text)
格式5: libcore.a:(.text)
格式6: :task.o(.text)
```

图 6.15

第一种格式的意思是指所有目标文件中的.text 段。目标文件可能来自于一个库，也可能是一个独立的文件。这种格式中使用了通配符以匹配目标文件的名称。图 6.16 示例说明了可以在链接脚本中使用的所有通配符。

通配符	含　义
*	匹配任意个数的字符
?	匹配任一字符
[_chars]	匹配_chars 所指定范围中的任一字符
\	脱字符。其功能类似于 C 语言中的“\n”中的反斜杠

图 6.16

第二种格式是指 data.o 目标文件中的.data 段。

第三种格式是指除了目标文件名满足 EXCLUDE_FILE 命令中的格式外所有目标文件中的.ctors 段。

第四种格式是指 libcore.a 库中 task.o 目标文件中的.text 段。

第五种格式是指 libcore.a 库中所有（目标文件）的.text 段。

第六种格式是指不在任一个库文件中的 task.o 目标文件中的.text 段。

从输入段描述的格式可以看出，括号内包含的是段名，而括号之前是描述目标文件名的字符或通配符。请注意，括号内的段名可以有多个，如图 6.17 所示。

```
00001: *(.text .rdata)
00002: *(.text) *(.rdata)
```

图 6.17

注意：上面两种格式并不等价。第一种格式中每一个目标文件中的.text 和.rdata 段是交替放在一起的；而第二种格式所生成的结果是各个目标文件中的.text 段放在一起，各个目标文件中的.rdata 段也是放在一起的，且整个.rdata 段放在整个.text 段的后面。

6.3 常用选项

除了 4.3.4 节介绍的-Map 选项可以让 ld 生成整个程序的映射文件外，还有其他几个值得注意的选项。

6.3.1 指定程序的入口点

通过使用 ld 的-e 选项，可以为可执行程序指定程序的入口点。它的功能与链接脚本中的 ENTRY 命令是一样的。

6.3.2 生成可重定位的中间文件

使用 ld 的-r 选项，可以将多个目标文件或多个库文件进行不完整的链接，且所生成的文件可以进一步被 ld 使用以便生成最终的可执行程序。图 6.18 示例说明了如何使用该选项。

```
/ClearRTOS/build
$ ld -r release/libcpu.a release/libcore.a -o libkernel.a
/ClearRTOS/build
$ ar x libkernel.a
ar: libkernel.a: File format not recognized
```

图 6.18

请注意，使用-r 选项后所生成的是一个已经经过部分链接的程序文件（即进行了部分重定位处理）。尽管上面的例子中使用了 libkernel.a 这样的名称，但是通过使用 ar 命令可以验证 libkernel.a 其实不是一个真正的库。

-r 选项的用处是，有时我们为了在自己的程序中生成所有程序的符号表数组，那么可以通

过两次链接的形式做到。第一次使用-r 选项将所有的目标文件和库进行链接，生成的输出文件通过使用 nm 等工具及另外的脚本生成符号表数组。最后对数据表数组进行编译以生成一个目标文件，并将之与第一次链接生成的文件再做一次链接以获得最终的可执行程序。

6.3.3　指定链接脚本

在嵌入式软件开发中，大多需要编写自己的链接脚本。通过使用-T 选项，可以覆盖 ld 所自带的链接脚本。



练习与思考

可执行程序与不可执行程序（如库、目标文件）的区别是什么？



第 7 章

gdb，程序调试助手

调试器（debugger）在软件开发中的作用无须多言。对于嵌入式软件开发工程师来说，熟练地以命令行的方式使用 gdb 进行软件调试是基本技能之一。本章将以 embedded 项目中的 mpool 示例程序为例，向读者展示如何使用 gdb 以命令行的方式进行软件调试。

首先运行“make debug”命令编译出调试版本的 embedded 项目，如图 7.1 所示。使用调试版本的原因是不希望编译器对程序进行优化，这使得生成的可执行程序与源程序间的对应关系非常清晰而有助于我们开展调试工作。

```
                    /embedded/build
$ make debug
                    /embedded/build
$ ls -l debug/mpool.exe
-rwxr-xr-x 1 yunli Domain Users 692K May 24 07:26 debug/mpool.exe*
```

图 7.1

7.1 启动和退出 gdb

一般有三种方式来启动 gdb。第一种方式如图 7.2 所示，这是开发阶段最常用的方式。这种方式与图 7.3 中启动 gdb 后再使用 file 命令是等价的。

```
                    /embedded/build
$ gdb debug/mpool.exe
```

图 7.2

```
                    /embedded/build
$ gdb
GNU gdb 6.8.0.20080328-cvs (cygwin-special)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-pc-cygwin".
(gdb) file debug/mpool.exe
Reading symbols from /embedded/build/debug/mpool.exe...done.
```

图 7.3

启动 gdb 的第二种方式如图 7.4 所示。在这种方式中，我们预先获得了被调试程序的一个 core 文件[①]core.pid。

```
                       /embedded/build
$ gdb debug/mpool.exe core.pid
```

图 7.4

启动 gdb 的第三种方式如图 7.5 所示。这是使用 gdb 调试已运行程序的启动方式，其中的 pid 是通过 ps 命令获得的进程号。

```
                       /embedded/build
$ gdb debug/mpool.exe pid
```

图 7.5

使用 quit 命令可以退出 gdb，如图 7.6 所示。

```
(gdb) quit

                       /embedded/build
```

图 7.6

7.2 获取帮助

gdb 通过提供完备的在线帮助，使我们使用起来更加方便。所有的帮助信息都是通过运行 help 命令获得的。运行 help 命令时，如果不指定参数，gdb 将输出分类帮助信息，如图 7.7 所示。

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

图 7.7

① core 文件是程序崩溃时所生成的内存转储文件，通过该文件可以还原程序崩溃时的情景。如果程序崩溃时没有生成 core 文件，可以运行“ulimit –c 50000”命令后再试试。

如果以分类名作为 help 命令的参数，则将获得哪些命令属于该分类，图 7.8 以栈分类为例示例了 help 命令的输出结果。

```
(gdb) help stack
Examining the stack.
The stack is made up of stack frames.  Gdb assigns numbers to stack frames
counting from zero for the innermost (currently executing) frame.

At any time gdb identifies one frame as the "selected" frame.
Variable lookups are done with respect to the selected frame.
When the program being debugged stops, gdb selects the innermost frame.
The commands below can be used to select other frames by number or address.

List of commands:

backtrace -- Print backtrace of all stack frames
bt -- Print backtrace of all stack frames
down -- Select and print stack frame called by this one
frame -- Select and print a stack frame
return -- Make selected stack frame return to its caller
select-frame -- Select a stack frame without printing anything
up -- Select and print stack frame that called this one

Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

图 7.8

通过这种渐进式的方式，我们能很快地获得需要使用的命令，进而以具体命令名为参数通过 help 命令获得更详细的帮助信息。另外有两点需要特别说明：在 gdb 中输入命令时，如果所输入的开头几个字符能唯一地标识该命令，则后面的字符就可以不用输入。以 run 命令为例，其简写形式就是一个 r。另外，在输入命令时可以使用 Tab 键让 gdb 自动为我们完成整个命令的输入，或者通过 Tab 键让 gdb 告诉我们哪些参数可选。为了表述方便，后面以“<Tab>”的形式表达在 gdb 中按下了 Tab 键。图 7.9 示例说明了调试 mpool.exe 程序时，在输入“mpool_”后按下 Tab 键所获得的效果。

```
(gdb) break mpool <Tab>
mpool_buffer_alloc    mpool_check_for_each   mpool_delete          mpool_dump_for_each
mpool_init            mpool_t                mpool_buffer_free     mpool_create
mpool_dump            mpool_handler_t        mpool_node_t
```

图 7.9

在 gdb 中，对于大多数命令都可以通过直接按 Enter 键的方式运行前面执行过的命令。图 7.10 示例说明了连续运行三个 next 命令的效果，其中的“<Enter>”代表回车键。

```
(gdb) next
41          if (module_registration_entry (argc, argv) != 0) {
(gdb) <Enter>
45          printf ("\nSystem is going to be up\n");
(gdb) <Enter>

System is going to be up
46          if (0 != system_up ()) {
```

图 7.10

7.3　调试程序

使用 gdb 调试程序需要有被调试程序的符号表。符号表可以是直接来源于被调试的程序文件，或者一个独立的符号表文件[②]。

7.3.1　断点设置

调试软件通常需要先设置好所需检查的程序点，即断点。在 1.9 节介绍了三种不同的断点：软件程序断点、硬件程序断点和数据断点，gdb 中设置这三种断点需要使用不同的命令。此外，gdb 还提供了捕获事件的断点设置命令。

7.3.1.1　软件程序断点

使用 break 命令可以设置软件程序断点。设置软件程序断点有多种形式，图 7.11 示例说明了设置断点的两种形式。第一种形式以函数名为 break 命令的参数，断点会设置在该函数的开始处；第二种形式通过指定文件名和行号的方式指定断点位置。当调用 break 命令不带参数时，gdb 会在程序指针寄存器所指的位置处设置一个断点。

```
(gdb) break mpool_create
Breakpoint 1 at 0x401603: file mpool.c, line 57.
(gdb) break mpool.c:147
Breakpoint 2 at 0x40197b: file mpool.c, line 147.
```

图 7.11

图 7.12 展示了使用 break 命令的另一种形式。我们可以使用 if 设置断点有效时应满足的先决条件，当表达式的值为非 0 时程序中断才发生。注意，表达式中既可以使用全局变量，也可以使用断点所在函数的局部变量。

```
(gdb) break mpool.c:85 if index = 3
```

图 7.12

当调试的是一个多线程程序时，可以在 break 命令中指定使断点有效的线程号。图 7.13 示例说明了 break 命令完整的语法格式。其中，方括号内的参数是可选的。

```
(gdb) break location [thread threadnum] [if condition]
```

图 7.13

如果希望设置一次有效的断点，可以使用 tbreak 命令。其使用方法与 break 命令完全相同。tbreak 是“temporary break”的简写。程序一旦在使用 tbreak 命令设置的断点处停下后，gdb 就会自动删除该断点。

② 使用 strip 命令的--only-keep-debug 选项，可以生成只包含符号表的文件。

使用 rbreak 命令可以以正则表达式的方式在多个函数中设置断点。图 7.14 展示了如何在所有以“mpool_”开头的函数内设置断点，其中一共在 8 处设置了断点。

```
(gdb) rbreak mpool_.
Breakpoint 1 at 0x40197a: file mpool.c, line 147.
void *mpool buffer alloc(mpool handler t);
Breakpoint 2 at 0x401a0c: file mpool.c, line 165.
error t mpool buffer free(mpool handler t, void *);
Breakpoint 3 at 0x401603: file mpool.c, line 57.
error t mpool create(const char *, mpool handler t *, void *, void *, msize t, msize t);
Breakpoint 4 at 0x401872: file mpool.c, line 117.
error t mpool delete(mpool handler t);
Breakpoint 5 at 0x401c35: file mpool.c, line 225.
void mpool dump();
Breakpoint 6 at 0x401d09: file mpool.c, line 246.
static bool mpool check for each(dll t *, dll node t *, void *);
Breakpoint 7 at 0x401ba0: file mpool.c, line 209.
static bool mpool dump for each(dll t *, dll node t *, void *);
Breakpoint 8 at 0x401596: file mpool.c, line 47.
static void mpool init();
```

图 7.14

设置断点时，gdb 会对每一个断点赋予一个唯一的标识值（后面我们称之为断点号）。通过使用“info breakpoints”命令可以查看已设置的（所有类型）断点，如图 7.15 所示。

```
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00401603 in mpool create at mpool.c:57
2       breakpoint     keep y   0x0040197b in mpool buffer alloc at mpool.c:147
```

图 7.15

使用 disable 和 enable 命令可以分别使任一个断点无效和有效，如图 7.16 所示。这两个命令的输入参数都是断点号，这两个命令会影响“info breakpoints”命令输出结果中“Enb”列的值。

```
(gdb) disable 1
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep n   0x00401603 in mpool create at mpool.c:57
2       breakpoint     keep y   0x0040197b in mpool buffer alloc at mpool.c:147
(gdb) enable 1
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00401603 in mpool create at mpool.c:57
2       breakpoint     keep y   0x0040197b in mpool buffer alloc at mpool.c:147
```

图 7.16

删除一个断点需要使用 delete 命令，图 7.17 示例说明了两种用法。提供断点号使得指定的断点被删除；使用“delete breakpoints”（无断点号参数）可以删除所有的断点。“delete breakpints *断点号*”命令的效果与“delete *断点号*”是一样的。

```
(gdb) delete 1
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x0040197b in mpool buffer alloc at mpool.c:147
(gdb) delete breakpoints
Delete all breakpoints? (y or n) y.
```

图 7.17

使用 ignore 命令可以让 gdb 忽略程序碰到某断点的次数，使用“help ignore”命令读者可以获得其使用帮助。

7.3.1.2 硬件程序断点

使用 hbreak 命令可以设置硬件程序断点，其命令格式与 break 是完全一样的。通过 show 命令可以了解处理器所支持的硬件断点数，如图 7.18 所示。

```
(gdb) show can-use-hw-watchpoints
Debugger's willingness to use watchpoint hardware is 1.
```

图 7.18

使用 thbreak 命令可以设置一次有效的硬件断点。

7.3.1.3 数据断点

数据断点的设置需要使用 watch 命令，其参数是我们所希望观察的被改变的变量名，或者是一个已知的内存地址。注意，如果想对局部变量使用 watch 命令，则需要程序已经停止在变量所在的函数内，对于全局变量就没有这一限制。读者可以想想这是为什么？此外，对于 32 位的处理器，数据断点只能用于监视类型大小为 32 位的数据。

7.3.1.4 事件断点

除了上面介绍的三类断点设置命令外，gdb 还提供了 catch 命令以便我们捕获调试期间的事件，事件包括信号、程序开始、程序终止和 C++中的异常（exception）等。读者可以通过 gdb 的在线帮助了解 catch 命令可用于捕获哪些事件。使用 tcatch 命令可以设置一次有效的事件断点。

7.3.2 控制程序运行

使用 run 命令可以运行被调试的程序。被调试程序所需的输入参数可以通过 run 命令加以指定。在 run 命令不带参数运行的情形下，以最后一次所指定的参数为准。

被调试程序的输入参数除了可以通过 run 命令加以指定外，还可以使用 set 命令设置 args 参数。图 7.19 示例说明了如何使用 set 和 show 命令来设置和显示 args 参数。

```
(gdb) set args -v -h
(gdb) show args
Argument list to give program being debugged when it is started is "-v -h".
```

图 7.19

除了 run 命令，start 命令也可用于启动被调试程序的运行，但程序会停在 main()函数的开始处，图 7.20 示例说明了其效果。这一命令等效于在 main()函数的入口处设置一个一次性断点，然后运行 run 命令。

```
(gdb) start
Breakpoint 2 at 0x401194: file /embedded/code/platform/common/inc/main.h, line 40.
Starting program: /embedded/build/debug/mpool.exe
[New thread 7356.0x1730]
[New thread 7356.0x1c24]
main (argc=1, argv=0x599f18) at /embedded/code/platform/common/inc/main.h:40
40      {
```

图 7.20

程序一旦停止（因为运行 start 命令或遇到断点）则可以使用 continue 命令让它继续运行。

在我们对程序进行单步跟踪时需要使用 next 命令。next 命令的后面可以带一个参数，指示命令的运行次数，如图 7.21 所示，即等价于我们连续运行三次 next 命令。

```
(gdb) next 3

System is going to be up
46          if (0 != system_up ()) {
```

图 7.21

如果要跟踪进入函数体内，则需要使用 step 命令。图 7.22 示例说明了在图 7.21 的基础上运行 step 命令的结果。很显然，step 命令使得程序进入了 system_up()函数。step 命令也可以像 next 命令那样携带需要运行的命令次数。

```
(gdb) step
system_up () at module.c:107
107         g_state = STATE_INITIALIZING;
```

图 7.22

跟踪代码时，我们可以使用不带参数的 list 命令（参见 7.3.3 节）让 gdb 显示跟踪点附近的代码，如图 7.23 所示。连续运行的 list 命令会接着上一次的显示结果列出源代码。

```
(gdb) list
102
103     error_t system_up ()
104     {
105         init_level_t level;
106
107         g_state = STATE_INITIALIZING;
108         for (level = LEVEL_FIRST; level <= LEVEL_LAST; ++ level) {
109             if (0 != dll_traverse (&g_levels [level], init_for_each,
110                 (void *)&level)) {
111                 return ERROR_T (ERROR_MODULE_INIT_FAILURE);
(gdb) <Enter>
112             }
113         }
114
115         g_state = STATE_UP;
```

```
116         for (level = LEVEL_FIRST; level <= LEVEL_LAST; ++ level) {
117             if (0 != dll_traverse (&g_levels [level], up_for_each,
118                 (void *)&level)) {
119                 return ERROR_T (ERROR_MODULE_UP_FAILURE);
120             }
121         }
```

图 7.23

从图 7.22 可知，此时程序停止在图 7.23 所显示代码的第 107 行，如果想让程序跳过后面的第一个 for 循环并停在第 115 行，则可以使用 until 命令，如图 7.24 所示。until 命令的参数是我们所希望的停止点，如果不带参数其效果与 next 命令是一样的。

```
(gdb) until 115
system_up () at module.c:115
115         g_state = STATE_UP;
```

图 7.24

如果想让程序运行完 system_up()函数并返回，则可以使用 finish 命令。图 7.25 示例说明了在 system_up()函数内使用该命令的效果。

```
(gdb) finish
Run till exit from #0  system_up () at module.c:115
0x004011d4          in          main          (argc=1,          argv=0x599f18)          at
/embedded/code/platform/common/inc/main.h:46
46          if (0 != system_up ()) {
Value returned is $1 = 0
```

图 7.25

如果希望在某函数内直接返回某一值而不真正运行完函数，则需要用到 return 命令。图 7.26 示例说明了在 system_up()函数中运行“return -1”的效果。

```
(gdb) return -1
Make system_up return now? (y or n) y
#0  0x004011d4 in main (argc=1, argv=0x599f18) at /embedded/code/platform/common/inc/
        main.h:46
46          if (0 != system_up ()) {
```

图 7.26

在 C 语言中如果想以汇编指令的方式调试程序，则需要用到 stepi 命令和 nexti 命令。图 7.27 示例说明了如何使用 display 命令（参见 7.3.3 节）让 gdb 每次停下来时打印出下一条需要执行的汇编指令，并演示了 nexti 命令和 stepi 命令的执行效果。

```
(gdb) display /i $pc
1: x/i $pc
0x4011d4 <main+[illegible]>:    test    eax, eax
(gdb) nexti
0x004011d6      46          if (0 != system_up ()) {
1: x/i $pc
0x4011d6 <main+[illegible]>:    je     0x40[illegible] <main+[illegible]>
(gdb) <Enter>
50          multitasking_start ();
1: x/i $pc
```

```
0x4011ed <main+109>:    call   0x402460 <multitasking start>
(gdb) stepi
multitasking start () at task.c:301
301     {
1: x/i $pc
0x402460 <multitasking start>:  push   %ebp
```

图 7.27

在调试过程中还可以人为地改变程序中变量的值，这需要用到 set 命令。该命令的格式是“set *变量名* = *表达式*”。

7.3.3　检查程序

调试程序时，如果要查看相关变量或寄存器中的值，则需要使用 print 命令。图 7.28 示例说明了 print 命令的格式。其中的 f 表示显示格式，可以是 x（十六进制）、d（有符号的十进制，这是默认格式）、u（无符号的十进制）、o（八进制）、t（二进制）、a（地址）、c（字符）和 f（浮点）中的任一个；expression 是指变量、函数等表达式。

```
(gdb) print /f expression
```

图 7.28

每一次执行 print 命令时，gdb 都会为输出结果分配一个编号，我们在之后可以使用“$n”的形式加以回访，其中的 n 就是 gdb 所分配的编号。图 7.29 展示了 gdb 的这一特性。

```
(gdb) start
Breakpoint 1 at 0x401194: file /embedded/code/platform/common/inc/main.h, line 40.
Starting program: /embedded/build/debug/mpool.exe
[New thread 2292.0x[illegible]]
[New thread 2292.0x13c4]
main (argc=1, argv=0x599f00) at /embedded/code/platform/common/inc/main.h:40
40      {
(gdb) print argc
$1 = 1
(gdb) print 1 + 1
$2 = 2
(gdb) print $1
$3 = 1
(gdb) print $2
$4 = 2
```

图 7.29

图 7.30 示例说明了如何通过 print 命令来检查某数据结构中各变量的值。

```
(gdb) break mpool create
Breakpoint 1 at 0x4[illegible]: file mpool.c, line 57.
(gdb) run
Starting program: /embedded/build/debug/mpool.exe
[New thread [illegible]]
[New thread [illegible]]
```

```
Breakpoint 1, mpool_create (name=0x40[illegible]214 "Test", p_handler=0x40d5c0, node=0x40c420,
    buffer=0x40[illegible], buffer_size=32, buffer_count=32) at mpool.c:57
57          mpool_node_t *p_node = node;
(gdb) next
58          address_t buffer_addr = (address_t) buffer;
(gdb) print p_node
$1 = (mpool_node_t *) 0x40c420
(gdb) print *p_node
$2 = {node_ = {prev_ = 0x0, next_ = 0x0}, addr_ = 0, in_use_ = 0}
```

图 7.30

另一个与 print 命令功能相似的命令是 call。call 命令与 print 命令的唯一区别是，当调用函数时如果某函数的返回值为 void，则它不打印出返回结果。图 7.31 示例说明了如何通过 call 函数来分配内存和对分配获得的内存进行初始化。

```
(gdb) call /x malloc (sizeof (int))
$5 = 0x599f98
(gdb) call memset (0x599f98, 'H', sizeof (int))
$6 = (void *) 0x599f98
```

图 7.31

如果希望检查某一内存区域中的数据，则需要使用 x 命令。x 命令的格式如图 7.32 所示。其中的 N 表示需要打印的单元个数；u 表示各单元的大小；而 f 则表示打印格式。u 可以是 b（字节）、h（双字节）、w（四字节）或 g（八字节）。f 除了包含 print 命令中的格式外，还可以是 s（以 null 为结束的字符串）或 I（汇编指令）。图 7.33 示例说明了如何使用 x 命令打印出图 7.31 所分配并初始化过的内存。

```
(gdb) x /Nuf expression
```

图 7.32

```
(gdb) x /4bc 0x599f98
0x599f98:       72 'H'  72 'H'  72 'H'  72 'H'
```

图 7.33

通过使用 info 和 print 命令可以查看寄存器的值，图 7.34 示例说明了如何使用它们。

```
(gdb) info reg
eax            0x40c420 4244512
ecx            0x401217 [illegible]
edx            0x0      0
ebx            0xdeaddead       -559030[illegible]
esp            0x40d564 0x40d564
ebp            0x40d59[illegible] 0x40d59[illegible]
esi            [illegible]
edi            [illegible]
eip            [illegible]
eflags         [illegible]
cs             [illegible]
ss             [illegible]
ds             [illegible]
es             [illegible]
```

```
fs             0x3b     59
gs             0x0      0
(gdb) info reg $ebp
$3 = (void *) 0x40d59c
(gdb) print $ebp
$3 = (void *) 0x40d59c
```

图 7.34

使用 backtrace 命令可以查看程序的调用栈，如图 7.35 所示。当程序因为崩溃而获得 core 文件时，我们可以通过使用 backtrace 命令了解程序的出错点（前提假设是栈没有被破坏）。backtrace 命令的输出结果为每一个栈帧（参见 10.4 节）提供了一个编号，这个编号可用于切换栈帧，后面我们会看到这一点。

```
(gdb) backtrace
#0  mpool_create (_name=0x409214 "Test", _p_handler=0x40d5c0, _node=0x40c420,
        _buffer=0x40c020, _buffer_size=32, _buffer_count=32) at mpool.c:58
#1  0x00401250 in task_test (_name=0x40ecfc "Test", _p_arg=0x0) at main.c:50
#2  0x00401d91 in task_main () at task.c:85
#3  0x004050f6 in root_context_init (_task_entry=0x401d5c <task_main>, _p_context=0x40ecbc)
      at context.c:48
#4  0x0040510e in root_frame_init (_p_frame=0x417e84) at context.c:55
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

图 7.35

使用“info locals”可以查看函数的所有局部变量的值，使用“info args”可以查看函数参数的值，使用“info frame”可以看到当前栈帧的信息，如图 7.36 所示。

```
(gdb) info locals
p_node = (mpool_node_t *) 0x40c420
buffer_addr = 0
level = 3735936685
handler = (mpool_handler_t) 0xdeaddead
index = 3735936685
initialized = 0
(gdb) info args
_name = 0x409214 "Test"
_p_handler = (mpool_handler_t *) 0x40d5c0
_node = (void *) 0x40c420
_buffer = (void *) 0x40c020
_buffer_size = 32
_buffer_count = 32
(gdb) info frame
Stack level 0, frame at 0x40d5a4:
 eip = 0x401[illegible] in mpool_create (mpool.c:58); saved eip 0x401250
 called by frame at 0x40d5c4
 source language c.
 Arglist at 0x40d59c, args: _name=0x409214 "Test", _p_handler=0x40d5c0, _node=0x40c420,
     _buffer=0x40c020, _buffer_size=32, _buffer_count=32
 Locals at 0x40d59c, Previous frame's sp is 0x40d5a4
 Saved registers:
  ebp at 0x40d59c, eip at 0x40d5a0
```

图 7.36

“info locals”和“info args”与栈帧是紧密相关的，通过使用 frame 命令我们可以在不同的栈

帧中进行切换。frame 命令的参数是 backtrace 命令所输出的各栈帧的编号。图 7.37 示例说明了将栈帧切换到 task_test()函数后，分别调用“info locals”、“info args”和“info frame”的输出结果。

```
(gdb) frame 1
#1  0x00401250 in task_test (_name=0x40ecfc "Test", _p_arg=0x0) at main.c:50
50          error = mpool_create ("Test", &handler, pool_node, pool_buffer,
(gdb) info locals
error = -559064411
p_buf1 = (void *) 0xdead5aa5
p_buf2 = (void *) 0xdead5aa5
p_buf3 = (void *) 0xdead5aa5
p_buf4 = (void *) 0xdead5aa5
p_buf5 = (void *) 0x0
handler = (mpool_handler_t) 0xdead5aa5
pool_node = {{node_ = {prev_ = 0x0, next_ = 0x0}, addr_ = 0, in_use_ = 0} <repeats 32 times>}
pool_buffer = {{buf_ = '\0' <repeats 31 times>} <repeats 32 times>}
(gdb) info args
 _name = 0x40ecfc "Test"
 _p_arg = (void *) 0x0
(gdb) info frame
Stack level 1, frame at 0x40d5e4:
 eip = 0x401250 in task_test (main.c:50); saved eip 0x401d91
 called by frame at 0x404604, caller of frame at 0x40d5a4
 source language c.
 Arglist at 0x40d5dc, args:  _name=0x40ecfc "Test",  _p_arg=0x0
 Locals at 0x40d5dc, Previous frame's sp is 0x40d5e4
 Saved registers:
  ebp at 0x40d5dc, eip at 0x40d5e0
```

图 7.37

另一种切换栈帧的方法是使用 up 命令和 down 命令。这两个命令的参数是一样的，用于指示希望向上或向下移动几个栈帧。

前面示例说明了如何使用 list 命令来查看源代码，图 7.38 示例说明了其他四种使用形式。在默认情形下，list 命令每次显示 10 行代码。“list -”命令用于显示前 10 行的代码。如果想更改 list 命令每次显示的行数，则可以使用 set 命令更改 listsize 参数的值。

```
(gdb) list mpool_create
50          }
51      }
52
53      error_t mpool_create (const char _name [], mpool_handler_t *_p_handler,
54          void *_p_node, void *_buffer, msize_t _buffer_size, msize_t _buffer_count
55      {
56          static bool initialized_ = false;
57          mpool_node_t *p_node = _node;
58          address_t buffer_addr = (address_t) _buffer;
59          interrupt_level_t level;
(gdb) list mpool.c:60
55      {
56          static bool initialized_ = false;
57          mpool_node_t *p_node = _node;
58          address_t buffer_addr = (address_t) _buffer;
59          interrupt_level_t level;
60          mpool_handler_t handler;
61          msize_t index;
```

```
62
63          if (is_in_interrupt ()) {
64              return ERROR_T (ERROR_MPOOL_CREATE_INVCONTEXT);
(gdb) list -
45          int idx;
46
47          dll_init (&g_free_mpool);
48          for (idx = 0; idx <= MPOOL_LAST_INDEX; idx ++) {
49              dll_push_tail (&g_free_mpool, &g_mpool_pool [idx].node );
50          }
51      }
52
53      error_t mpool_create (const char name [], mpool_handler_t * p_handler,
54          void * node, void * buffer, msize_t buffer_size, msize_t buffer_count)
(gdb) list 53,70
53      error_t mpool_create (const char name [], mpool_handler_t * p_handler,
54          void * node, void * buffer, msize_t buffer_size, msize_t buffer_count)
55      {
56          static bool initialized = false;
57          mpool_node_t *p_node = node;
58          address_t buffer_addr = (address_t) buffer;
59          interrupt_level_t level;
60          mpool_handler_t handler;
61          usize_t index;
62
63          if (is_in_interrupt ()) {
64              return ERROR_T (ERROR_MPOOL_CREATE_INVCONTEXT);
65          }
66          if (null == node || null == buffer) {
67              return ERROR_T (ERROR_MPOOL_CREATE_INVPTR);
68          }
69
70          level = global_interrupt_disable ();
```

图 7.38

如果需要查看反汇编代码，则可以使用 disassemble 命令。在不带参数的情形下，disassemble 显示程序计数器寄存器所指向地址附近的汇编代码。如果给 disassemble 指定程序地址，它将显示该地址附近的汇编代码。

7.3.4　提高调试效率

调试效率的提高除了正确设置断点外，还需要使用 gdb 的其他功能。第一个可以考虑的功能是使用 display 命令，其命令格式如图 7.39 所示。该命令的参数与 print 命令是相同的。

```
(gdb) display /f expression
```

图 7.39

使用 display 命令所设置的表达式在程序每次碰到断点停止时都会自动打印出其值。在前面我们示例说明了如何使用 display 命令来显示下一条将要运行的汇编指令。通过这种方式，可以省去每次手工检查的麻烦。使用 display 命令不带参数时，将获得已设置了哪些自动显示内容；去除自动显示内容需要使用 undisplay 命令。图 7.40 示例说明了这两个命令的使用方法。

```
(gdb) display /i $pc
1: x/i $pc
0x401250 <task_test+57>:        mov     eax,-0x18(%ebp)
(gdb) display
1: x/i $pc
0x401250 <task_test+57>:        mov     eax,-0x18(%ebp)
(gdb) undisplay 1
(gdb) display
(gdb)
```

图 7.40

提高调试效率的第二种方式是，可以通过在 gdb 中编辑和编译程序的方式，省去启动和退出 gdb 的动作。我们可以使用 edit 命令编辑源程序，edit 命令的参数可以是文件、函数名和行号；也可以运行 cd、pwd 和 make 命令进行工作目录切换和程序编译。gdb 一旦发现被调试程序有更新，就会自动地重新加载程序（和符号表）。此外，如果需要在 gdb 中运行其他的 shell 命令，则可以使用 gdb 中的 shell 命令。图 7.41 示例说明了如何在 gdb 中调用 shell 中的 date 命令。

```
(gdb) shell date
Wed Jun  1 15:12:19 CST 2011
```

图 7.41

在有的操作系统中，当 gdb 正在调试一个程序时，程序文件将会被锁定而使得无法进行编译，此时需要使用 kill 命令关闭调试程序。

7.4　查看符号表

图 7.42 示例说明了如何通过 info、ptype 和 whatis 三个命令来查看程序中的符号表。

```
(gdb) info variables g_mpool_pool
All variables matching regular expression "g_mpool_pool":

File mpool.c:
static mpool_t g_mpool_pool[2];
(gdb) ptype g_mpool_pool
type = struct {
    dll_node_t node_;
    magic_number_t magic_number_;
    char name_[16];
    address_t addr_start_;
    address_t addr_end_;
    msize_t buffer_size_;
    msize_t buffer_count_;
    int buffer_size_in_bits_;
    bool apply_shift_;
    dll_t free_buffer_;
    mpool_node_t *p_node_;
    statistic_t stats_nobuf_;
} [2]
(gdb) info address mpool_create
Symbol "mpool_create" is a function at address [illegible].
(gdb) info functions mpool_create
All functions matching regular expression "mpool_create":
```

```
File mpool.c:
error_t mpool_create(const char *, mpool_handler_t *, void *, void *, msize_t, msize_t);
(gdb) whatis mpool_create
type = error_t (const char *, mpool_handler_t *, void *, void *, msize_t, msize_t)
```

图 7.42

7.5 控制 gdb 的行为

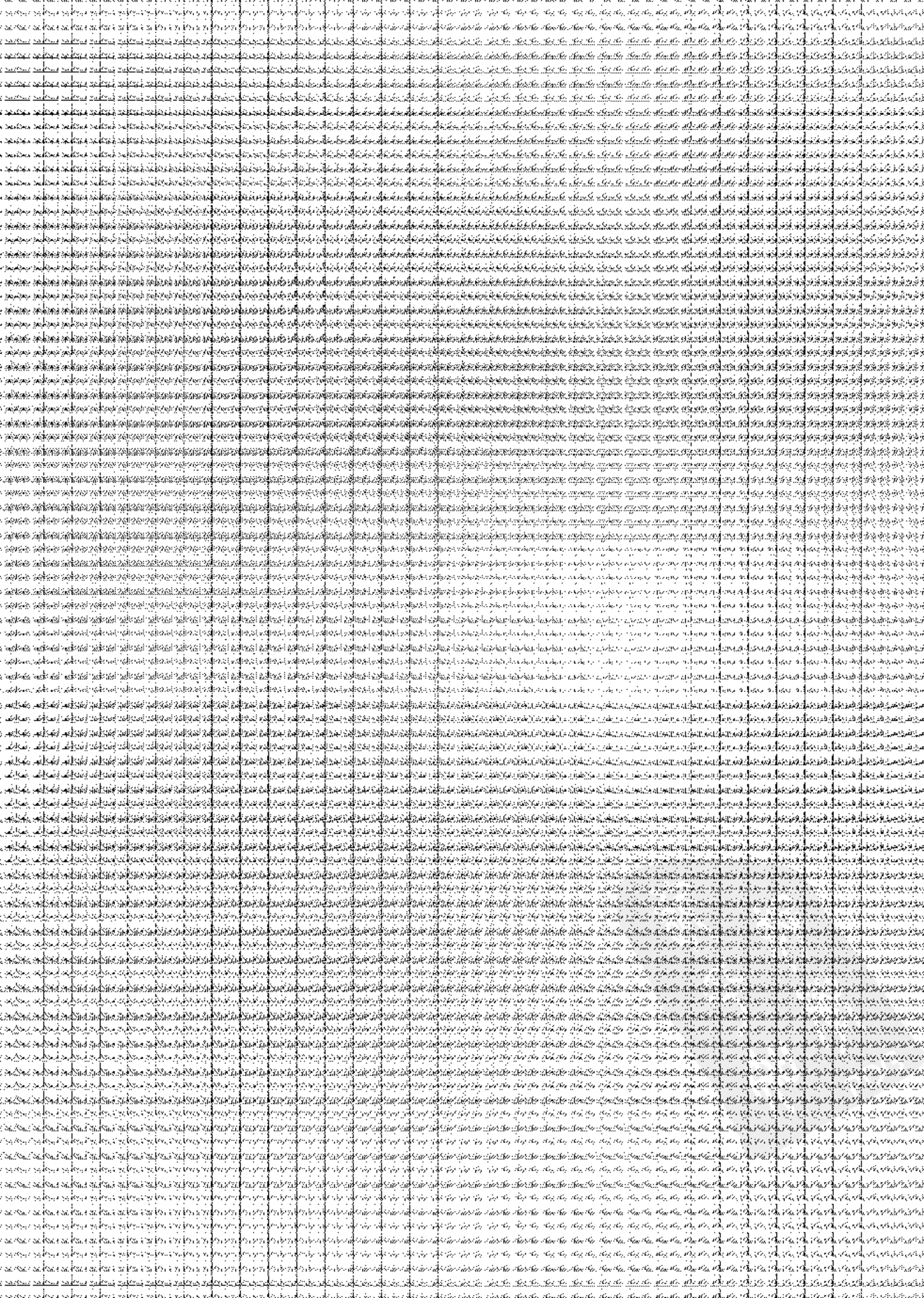
gdb 通过提供参数的形式，让我们能控制其行为。通过使用 set 和 show 命令，可以设置和查看各参数。图 7.43 以 listsize 参数为例，看它是如何改变 list 命令的每次显示行数的。

```
(gdb) show listsize
Number of source lines gdb will list by default is 10.
(gdb) set listsize 15
(gdb) list
35      #endif
36
37      int module_registration_entry (int argc, char *argv []);
38
39      int main (int argc, char *argv [])
40      {
41          if (module_registration_entry (argc, argv) != 0) {
42              printf ("Error: module registration failure\n");
43              return -1;
44          }
45          printf ("\nSystem is going to be up\n");
46          if (0 != system_up ()) {
47              printf ("Error: system cannot be up\n");
48              return -1;
49          }
```

图 7.43

通过“help show”命令可以了解在 gdb 中有哪些参数。





编程语言篇



设置本篇的目的并不是对 C 语言语法知识的重复介绍，而是希望读者通过阅读本篇对 C 语言的理解能上一个台阶，这对于一个专业的嵌入式软件开发工程师来说是必须的。

全面深入的嵌入式软件开发一定离不开编写汇编程序。不管是从头开始编写一个纯粹的汇编源文件，还是使用嵌入汇编的方式，都需要我们掌握汇编器的语法。第 8 章的设置正是为了这一目的。由于汇编程序在项目中的比重比较低(个人认为低于 5%)，因此该章的篇幅也相对较小，作者编写该章时是立足于让读者能理解本书所涉及的汇编知识的。

所编写的 C 程序被编译器编译成的可执行文件是什么样的？栈和堆，是 C 语言中两个非常重要的概念，它们的作用是什么？我们该怎样来管理它？在第 9 章通过深入地探讨程序中的段、栈和堆，来帮助读者理解这些问题。

编译器将 C 语言转换成处理器指令的依据是什么？它是如何分配处理器的寄存器的？它又是如何处理数据的边界对齐问题的？处理器各寄存器的功能是如何分配的？C 语言中的函数参数是如何传递的？局部变量的内存又是如何分配的？什么是栈帧？等等。这一系列的问题都能从 ABI/EABI 规范中找到答案，这是第 10 章的主题。

C 语言中的指针是一大难点。在第 11 章中将通过探究一个因混淆指针类型所造成的奇怪问题的根源来深入地了解数组和指针的内存模型，并最终得出我们应当养成“总是将头文件作为(变量和函数的)定义和引用的桥梁”这一编程好习惯。

嵌入式软件开发的一个显著特点是需要与硬件打交道，也离不开编写硬件驱动程序这一工作内容。编写硬件驱动程序很可能需要用到 C 语言中的 volatile 关键字，第 12 章就 volatile 关键字的用处进行了探索。



第 8 章

掌握必要的汇编知识

gcc 中的 as 是专门用于对汇编代码进行编译的工具，通常称之为汇编器。本章并不打算就 as 的选项进行讲解，而是关注于介绍阅读本书所需的汇编语法知识，读者可以运行“man as”命令获得其帮助信息。

第 20 章讲解 ClearRTOS 操作系统的任务实现时，需要通过编写汇编程序以实现任务情景的切换，本章内容将围绕这些汇编代码展开，以便读者能更好地消化这部分内容。

8.1 as 的语法

为了读者阅读方便，20.1.3 节将要介绍的任务情景恢复函数 context_restore()的实现被拷贝到了本章，如图 8.1 所示。

```
00060: void context_restore (task_context_t *_p_context, int _value);
00061:
```

```
00026: #include "offset.h"
00027:
00028:     .globl _context_restore
00029:     .globl context_restore
00030:     .align 4
00031:
00032: _context_restore:
00033: context_restore:
00034:     // get _p_context of context_restore ()
00035:     movl FRAME_OFFSET_PARAM0(%esp), %ecx
00036:
00037:     // take 2nd parameter of context_restore () as return value of context_save ()
00038:     movl FRAME_OFFSET_PARAM1(%esp), %eax
00039:
00040:     // restore registers from context
00041:     movl (CONTEXT_OFFSET_EIP*4)(%ecx), %edx
00042:     movl (CONTEXT_OFFSET_EBX*4)(%ecx), %ebx
00043:     movl (CONTEXT_OFFSET_ESI*4)(%ecx), %esi
00044:     movl (CONTEXT_OFFSET_EDI*4)(%ecx), %edi
00045:     movl (CONTEXT_OFFSET_EBP*4)(%ecx), %ebp
00046:     movl (CONTEXT_OFFSET_ESP*4)(%ecx), %esp
```

```
00047:
00048:     // jump to context_save ()
00049:     jmp *%edx
```

图 8.1

context.h 中示例说明了 context_restore()函数的原型，而其汇编实现位于 restore.S 文件中。接下来围绕 restore.S 文件展开介绍。

8.1.1 宏

从 restore.S 文件的第 26 行可以看到我们熟悉的、在 C 语言中经常使用的#include 宏指令。是的，as 也支持宏。但是，汇编程序中所包含的头文件不能包含 C 语言中的函数、数据结构等内容，而只能定义常量、汇编指令等汇编器认识的内容。

宏的存在说明 as 在编译时存在预处理这一步骤，这一点与编译 C 程序时的预处理是相似的。但读者需要注意，如果所编写的汇编代码中包含宏指令的话，则一定要将源文件的后缀名使用大写的 S。对于图 8.1 中的汇编文件，如果文件名为 restore.s 的话，as 会报告说它不认识#include 指令。

8.1.2 汇编命令

汇编命令都是以“.”开头的，用于指示 as 如何处理我们所写的汇编代码。比如，图 8.1 中第 28 和 29 行的.globl 命令使得跟在其后的符号在文件之外可见，或者从 C 语言的角度来看，类似于定义全局函数。第 30 行的.align 命令是告诉 as 后面代码的存放地址必须满足 4 字节边界对齐要求。

as 支持大量的汇编命令，这些命令都可以从 as 的官方手册《Using as》（附书光盘中有，其文件名为 as.pdf）中找到。在这些汇编命令中有几个值得在此一提。第一批包括.text 和.data 命令，它们指示将其后的内容放入指定段中；第二批是.lcomm 和.comm，它们指示将其后的内容放入程序的.bss 段中。关于段的知识下章会详细讲解。

8.1.3 符号和标签

符号（symbol）或许是程序中非常核心的概念，我们通过符号去表达变量和命名函数，这一点在汇编程序中也与此相似。图 8.1 的第 28 和 29 行就声明了两个后面将要定义的符号，它们其实对应于 C 语言中的函数名。

符号可以由字母、“.”和“_”组成，对于有的处理器还可以使用“$”。

符号后面如果加上一个“:”就成为了标签（label），比如图 8.1 中的第 32 和 33 行就是两个标签，标签中的符号其实代表的是程序运行时在内存中的一个具体地址。从这一点来看，符号是有值的。标签与 C 语言中的函数是对等的。

8.1.4　汇编指令

图 8.1 的第 34～49 行是 context_restore()函数的汇编指令实现。不同的处理器具有不同的指令集，而这也造成所编写的汇编代码中的指令完全不同。这段代码只使用到了 x86 处理器中的 mov 和 jmp 两个指令，具体指令的含义读者可以查看附书光盘中的指令手册，在此不展开细讲。

一旦读者参考附书光盘中 x86 处理器的指令手册就会发现，手册中指令的格式与图 8.1 所列的有所不同。这种不同是由存在 AT&T 和 Intel 两种语法格式而造成的，图 8.2 中的表列出了它们之间的部分区别。不论是 AT&T 语法还是 Intel 语法 as 都支持，可以通过.att_syntax 和.intel_syntax 汇编命令来切换指令格式。

由于 AT&T 格式中操作数内存大小是通过在汇编指令后面加上一个字母来表示的，因此读者在查看 x86 处理器的指令手册时需要根据情况除去后面的那个字母，否则无论如何在手册中也找不到对应的指令。后缀字母“b(byte)”、“w(word)”、“l(long)”、“q(quardruple)”分别表示单字节、双字节、四字节和八字节。

差　异　点	AT&T 语法格式	Intel 语法格式
寄存器操作数的表达	总是在寄存器名前加一个“%”	
常数和立即数的表达	`pushl $4`	`push 4`
操作数的源与目的的表达	`addl $4, %eax`	`add eax, 4`
操作数内存大小的表达	`movb foo, %al`	`mov al, byte ptr foo`
内存引用方式	`movl -4(%esp), %ecx` `movl 4(%esp), %ecx`	`mov ecx, [esp-4]` `mov ecx, [esp+4]`

图 8.2

图 8.1 的第 49 行的 jmp 指令中使用了一个“*”，它表示绝对跳转。如果没有这一符号，则表示相对跳转。

8.2　嵌入汇编的语法

20.4.1 节在介绍任务创建函数的程序实现时涉及了 root_frame_init()函数，这个函数被用于创建任务的初始栈帧（参见 10.4.1 节），函数的实现使用了在 C 程序中嵌入汇编的方式。同样地，为了阅读方便，将函数的实现内容拷贝到了图 8.3 中。

```
00053: static void root_frame_init (root_frame_t *_p_frame)
00054: {
00055: asm volatile(
00056:     // save old ESP into root_frame_t
00057:     "movl %%esp, 0x14(%0) \n"
```

```
00058:      // switch to new stack which starts from _p_frame
00059:      "movl %0, %%esp \n"
00060:      // initialize EBX
00061:      "popl %%ebx \n"
00062:      // initialize ESI
00063:      "popl %%esi \n"
00064:      // initialize EDI
00065:      "popl %%edi \n"
00066:      // The reason why we don't want to initialize the EBP is when the code is build
00067:      // with -O2 option, the GCC will optimize the code to use EBP at the end of
00068:      // root_context_init (). So if we initialize the EBP here it will cause crash.
00069:      "call _root_context_init \n"
00070:      "movl 8(%%esp), %%esp \n"
00071:      ::"r"(_p_frame):"ebx","esi","edi"
00072:      );
00073: }
```

图 8.3

嵌入汇编的格式是：

```
asm volatile("汇编指令"
              : 输出寄存器(列表)
              : 输入寄存器(列表)
              : 保留寄存器(列表));
```

其中的 volatile 用于防止编译器对嵌入的汇编指令进行优化，它是可选的。

从图 8.3 的第 57 行可以看出，寄存器名前都得加上“%%”前缀，这一点与非嵌入汇编是不一样的。另外，还可以看到像“%0”这样奇怪的符号，后面我们再介绍它是什么意思。除了这两点特殊之处外，嵌入的汇编指令与非嵌入的汇编指令是完全一样的。由于所有的汇编指令是写在一起的，所以必须用“\n”对每一条汇编指令进行分割，这一点读者也需注意。

嵌入的汇编代码在大多情形下需要使用到 C 程序中定义的变量或函数参数，因此在嵌入的汇编内需要完成寄存器与变量和参数的映射，这正是嵌入汇编格式内“输入寄存器”和“输出寄存器”的作用。从 C 语言的角度来看，“输入寄存器”用于指定输入变量（或参数，后同）与处理器寄存器的映射关系，而“输出寄存器”则用于指定输出变量与寄存器的映射关系，这里的出与入是从被嵌入的汇编角度来理解的。

“输入寄存器”的格式必须是：

```
"限制"(C变量或参数名)
```

其中的“限制”是一些字母，这些字母用于指明处理器的一个寄存器，这个寄存器将用于指代括号内的 C 程序名称。对于 x86 处理器，“限制”可以使用的字母如图 8.4 所示。如果有多个输入变量需要指定，则各部分需用“,”加以分割。

“输出寄存器”的格式必须是：

```
"=限制"(C变量或参数名)
```

除了在“限制”之前多了一个等号外，输出寄存器与输入寄存器的格式是一样的。另外，

由于输出寄存器的值将放入其后括号内的 C 程序符号中，因此它必须是左值。在图 8.3 中，由于汇编部分不需要输出结果到 C 程序中，所以“输入寄存器”部分是空的。

限制	寄存器分配含义
a	指定 eax
b	指定 ebx
c	指定 ecx
d	指定 edx
S	指定 esi
D	指定 edi
I	从 0~31 的常量
q	从 eax、ebx、ecx、edx 中动态分配一个
r	从 esi 和 edi 中动态分配一个
g	从 eax、ebx、ecx、edx 或内存变量中动态分配一个
A	指定 eax 和 edx 合成一个用于存储 64 位的数(long long)

图 8.4

当使用“q”、“r”和“g”这些限制时，表示由编译器自动完成寄存器的分配。那如何在嵌入的汇编中引用被编译器自动分配的寄存器呢？这得通过“%N”这样的格式加以引用的，其中的 N 可以从 0 到 9，N 指示了寄存器在“输出寄存器”和“输入寄存器”列表中的索引。如果输入和输出寄存器都有两个，%0 代表第一个输出寄存器，%1 代表第二个输出寄存器，%2 代表第一个输入寄存器，而%3 代表第二个输入寄存器。当没有输入寄存器时，%0 就代表第一个输入寄存器。回头看一看图 8.3 中的第 57 行，读者不难理解其中的%0 其实就是指代表 _p_frame 参数的寄存器，且这一寄存器是由编译器从 esi 和 edi 中分配而来的。

“保留寄存器”用于告诉编译器不要为“输入寄存器”和“输出寄存器”分配这些寄存器。

在后面的 20.4.1 节中读者将进一步了解图 8.3 中的嵌入汇编最终完全转换成汇编的模样。



第 9 章 深入理解程序的结构

本章的程序特指运行在处理器上的指令和指令所加工的数据，这些指令和数据来源于程序文件。

程序是被分成段（section）加以管理的。在程序被加载到内存中运行之前，各段是放在程序文件中的。在 5.4 节中，介绍了如何使用 objdump 工具来查看一个程序文件中的段信息，第 19 章会介绍引导加载器是如何加载一个程序的。

当程序文件中所必需的段被加载到内存中后，将通过运行指令来处理相应的数据。有些数据来源于程序文件中的段，有些则是动态生成的。动态生成的数据可以来自栈（stack）或堆（heap）[①]。

通过掌握程序的段、栈和堆有助于深入地理解 C 编程语言，使得我们在编程活动中能准确地使用各种定义变量的方法，以及分析各种方法所带来的潜在影响。

9.1 段

传统上，一个程序一般会有这几个段：.text、.data 和.bss 段。下面就来说一说各段的作用是什么，以此了解 C 语言中的各元素（函数和变量）是被放到哪一个段中的。

9.1.1 指令段

先来说一说.text 段。在 1.5 节谈到，处理器只认识指令与数据。不论是采用什么高级语言编写的程序，其最后都得被编译器转换为处理器所认识的机器指令和数据，这个转换过程正是我们熟悉的“编译”。而编译所生成的指令，就是存放在.text 段中的。从 C 源程序的角度来看，.text 中存放的是函数的机器指令实现。

如果处理器有内存管理单元，那么可执行程序被加载到内存以后，通常会将.text 段所在的内存空间设置为只读，以保护.text 中的代码不会因为程序出错（比如，非法指针等）而被意外

① 有些书中将“stack”翻译成“堆栈”，作者觉得更好的翻译是“栈”，因为“堆”在 C 语言中具有特殊的含义。

地改写。

9.1.2 数据段

处理器所需加工的数据是放在.data、.bss 和.rdata 段的。当然，除这几个段外，数据也可以来自栈和堆，这部分内容在后面我们会继续探讨。下面通过例子来了解各段中放的是 C 语言中什么类型的数据。让我们以图 9.1 所示的程序作为基准，来了解.data 和.bss 两个段中所存放数据的区别。

```
section1.c
int main ()
{
   return 0;
}
```

图 9.1

图 9.2 示例说明了编译生成的目标文件用 objdump 所看到的段信息。

```
$ gcc -g -c section1.c

$ strip section1.o

$ objdump -h section1.o

section1.o:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000024  00000000  00000000  0000008c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000000  2**2
                  ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000000  2**2
                  ALLOC
```

图 9.2

在输出的 objdump 信息中，请注意其中用下划线所标出来的两个段的大小。现在，在 section1.c 中定义两个全局变量，更改后的代码如图 9.3 所示。新增的两个变量都是初始化好的，其中一个初始化为 0。

```
int g_nonzero = 0x5A5A5A5A;
int g_zero = 0;

int main ()
{
   return 0;
}
```

图 9.3

图 9.4 示例说明了新程序使用 objdump 所获得的信息。

```
$ gcc -g -c section2.c
$ strip section2.o
$ objdump -h section2.o

section2.o:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000024  00000000  00000000  0000008c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  00000000  00000000  000000b0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  00000000  00000000  00000000  2**2
                  ALLOC
```

图 9.4

与图 9.2 所显示信息不同的是，.data 段和.bss 段的大小有了改变。这是因为在 section2.c 中增加了两个变量。对于初始化不为 0 的变量，编译器会将它放入.data 段中，而对于初始化为 0 的变量会被放入.bss 段中。实际上，没有初始化的变量也像初始化为 0 的变量那样被放入.bss 段中。

通过 objdump，可以看到.data 段对应变量的值，如图 9.5 所示。对于这一输出信息，需要注意处理器的 endian 模式，有可能所看到的信息与在源程序中定义的字节序是相反的。

```
$ objdump -s -j .data section2.o

section2.o:     file format pei-i386

Contents of section .data:
 402000 00000000 00000000 00000000 5a5a5a5a  ............ZZZZ
```

图 9.5

由于.bss 段中存放的数据是初始化为 0 或没有初始化好的，所以不需要将其内容像.data 段那样存放在程序文件中，可以通过查看程序文件中.bss 段中的内容来加以验证，如图 9.6 所示。

```
$ objdump -s -j .bss section2.o

section2.exe:     file format pei-i386
```

图 9.6

从显示结果来看，程序文件中.bss 段的内容的确是空的。当程序被引导加载器加载以后，引导加载器将执行权交给被加载程序之前，它会把.bss 段所在的内存区全部初始化为 0。这就是为什么没有设置初始化值的全局变量其值却总是为 0 的缘故。注意，不要误以为未初始化的

局部变量（不是全局变量）其值也会被自动初始化为 0。

由于.bss 段中的变量不需要初始化成特定值（0 除外），所以不需要在程序文件中保存其内容，其好处是能减小程序文件的大小而节省存储空间。

置于.data 段内数据的初始化，是引导加载器加载程序时，通过将程序文件中.data 段的数据复制到所对应的内存地址空间，从而一次性地完成所有变量的初始化。

通过 nm 工具，也可以验证两个变量所分配的段信息[②]。验证结果如图 9.7 所示。

```
$ nm -n section2.o
...... 显示结果有删减 ......
00000000 D g_nonzero
00000000 B g_zero
```

图 9.7

在一个函数内定义非静态变量（即局部变量）时，变量的内存空间是被分配在栈上的，那加了 static 后还分配在栈上吗？我们需要通过实验来探个究竟。图 9.8 所示是实验需要用到的源程序，objdump 的输出结果在图 9.9 中给出。

```
int main ()
{
   static int s_nonzero = 0x5A5A5A5A;
   static int s_zero = 0;
   return 0;
}
```

图 9.8

```
$ gcc -g -c section4.c

$ strip section4.o

$ objdump -h section4.o

section4.o:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         [illegible]  [illegible]  [illegible]  [illegible]  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  [illegible]  [illegible]  [illegible]  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000010  [illegible]  [illegible]  [illegible]  [illegible]
                  ALLOC
```

图 9.9

② 注意，使用 nm 之前不能对程序文件使用 strip 命令。

与图 9.4 相类比会发现，对于静态的局部变量，编译器为之分配的内存空间与全局变量是一样的。细心的读者会发现，.bss 段的大小是 16 个字节，而非之前的 4 个字节，至于为什么，留给读者去探索。

前面讲了 int 类型的全局变量，下面再来看一看字符串全局变量。同样地，需要一个新的测试程序来帮助探究，如图 9.10 所示。

```
char g_char [] = "Hello World!";

int main ()
{
   return 0;
}
```

图 9.10

图 9.11 所示是编译及段显示结果。从图中可以看出，g_char 变量是被分配在.data 段内的。其分配的长度是 16 个字节，而不是实际的 13 个字节，原因是为了满足段的 4 字节对齐要求。

```
$ gcc -g -c section5.c

$ strip section5.o

$ objdump -h section5.o

section5.o:     file format pei-i386

Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000024  00000000  00000000  0000008c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000010  00000000  00000000  000000b0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000000  2**2
                  ALLOC

$ objdump -s -j .data section5.o

section5.o:     file format pei-i386

Contents of section .data:
 0000 48656c6c 6f20576f 726c6421 00000000  Hello World!....
```

图 9.11

在 g_char 变量前加 const 关键字，改变后的源程序如图 9.12 所示。图 9.13 所示是其编译及段显示结果。这次与 section5.exe 的结果有所不同，是.rdata 段的大小发生了改变。

```
const char g_char [] = "Hello World!";

int main ()
{
   return 0;
}
```

图 9.12

```
$ gcc -g -c section6.c

$ strip section6.o

$ objdump -h section6.o

section6.o:     file format pei-i386

Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000024  00000000  00000000  000000b4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000000  2**2
                  ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000000  2**2
                  ALLOC
  3 .rdata        00000010  00000000  00000000  000000d8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA

$ objdump -s -j .rdata section6.o

section6.o:     file format pei-i386

Contents of section .rdata:
0000 48656c6c 6f20576f 726c6421 00000000  Hello World!....
```

图 9.13

.rdata 是用来存放只读的已初始化变量的。当在源程序中的 g_char 变量前面加了 const 后编译器就知道这个字符串是只读的，所以将其分配到.rdata 段中。与.data 段不同的是，对于有内存管理单元的系统，.rdata 段通常也会采用内存管理单元进行只读保护。

除了.text、.data 和.bss 三个段外，在程序中还可能存在其他的段。不管各段的名称有多么不同，但都逃不出它们只能属于指令、数据和调试信息三个类别。属于调试信息的段只被调试工具所使用，与程序的实际运行毫无关系。

9.2 栈

栈的作用包括：

- 当中断发生时用于保存处理器寄存器的值，以便中断返回时通过退栈回到被中断点继续程序的运行。
- 用做函数参数和局部变量的存储空间。正因为栈还可以用于存放局部变量，所以栈指针的变化并不只是通过压栈和退栈才可以改变。实际上，编译器会根据我们所编写函数内局部变量的大小调整栈的指针。10.4.1 节就栈的作用会做进一步的介绍。

一旦一块内存空间成为了栈后，栈操作就是通过处理器的栈指针寄存器（后面简称栈指针或 SP）来完成的。

通过栈指针，可以实现压栈和退栈两种不同的操作。栈存在方向性，是指当进行一次压栈操作时，栈指针是向更高地址处变化，还是反过来向更低地址处变化。在 x86 处理器上，当进行压栈操作时，栈指针是向低地址变化的。图 9.14 示例说明了在 x86 处理器上进行栈操作时栈指针的变化情形。

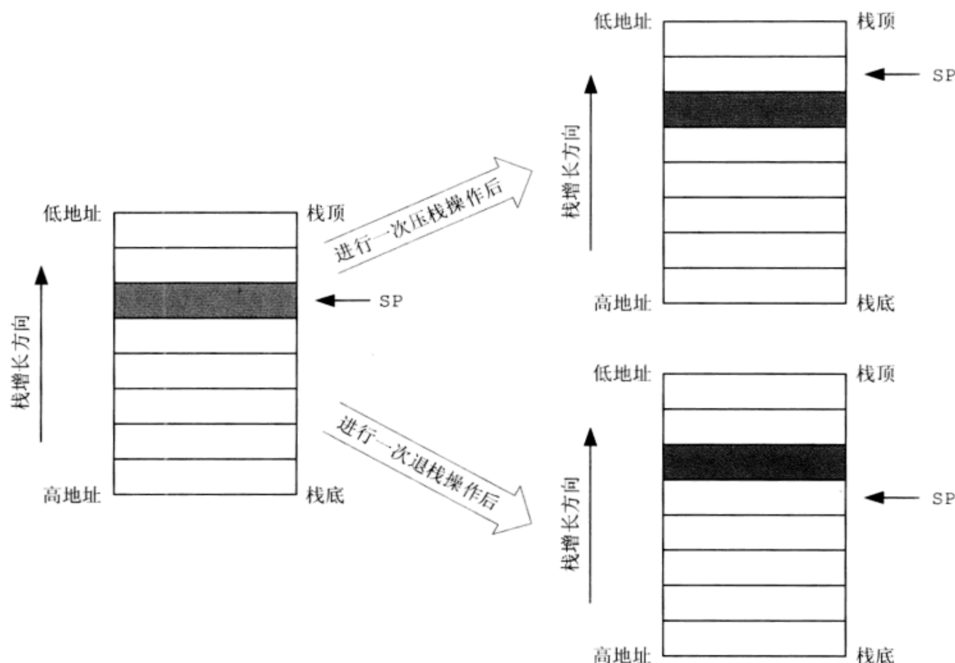


图 9.14

一次栈操作意味着栈指针移动一定数量的字节，对于 32 位处理器来说，一次操作将造成栈指针移动 4 个字节。

压栈操作是将数据放入栈中。比如进行一次寄存器的压栈操作后，寄存器的值将被保存到图 9.14 中右上角 SP 所指向的内存空间中。压栈操作的具体顺序是，先更新 SP 的值，然后将寄存器的值放入 SP 所指向的栈空间中。相反，如果是进行一次退栈操作，退栈前 SP 所指向的内存空间中的值会被放入指定的寄存器中。比如图 9.14 中带阴影部分内存中的值在退栈时将被放入寄存器中。退栈的具体动作是，先将 SP 所指向栈空间中的值放入寄存器中，然后调整 SP。

从图中可以看出，栈只有一边是可以变化的，变化是通过 SP 来实现的。对于可变化的一端被称为栈顶，不可变化的那一端被称为栈底。显然，栈具有先进后出的特性。

由于栈被运用于实现函数调用，而在多任务环境中，各任务的函数调用路径可以不同，所以每个任务都需要有自己独立的栈空间。任务的栈空间及大小是在任务创建时指定的，在第 20 章讲解任务时，还将就任务与栈的关系进行介绍。

在函数中定义局部变量将导致变量所占用的内存被自动分配在栈上，这一分配工作是由编译器在编译时生成的指令自动完成的。函数的返回意味着其局部变量所占用的栈空间被自动释放，释放操作同样是由编译器在编译时产生的指令自动完成的。在下一章介绍栈帧时，相信读者能更好地理解这一点。

9.3 堆

在第 19 章中将谈到，当引导加载器加载完了应用程序以后，单体应用程序在操作系统的初始化阶段也需要预先准备好一块内存区域为函数的调用提供初始栈空间，在完成了基本的初始化以后，将形成图 9.15 所示的内存布局。



图 9.15

图中的闲置内存空间就会被用做整个系统的堆空间。堆的作用是为整个系统提供动态分配内存的空间，C 语言库中的 malloc()函数就是从堆中获取内存的。对于堆的管理，操作系统中存在专门的一个内存管理模块（参见第 22 章）。当内存管理模块在初始化时，需要知道它所管理的内存空间的起始地址分别是什么，这需要通过链接脚本的配合来获取。图 6.5 的链接脚本内的__end__符号就表示了堆的开始地址，堆的结束地址还是容易获得的，因为每一个嵌入式系统都可以事先知道该系统的最大内存容量。

从堆中获取内存需要调用像 malloc()这样的函数，而使用完了后又要调用像 free()这样的函数释放内存。对于不再使用的内存，如果忘记进行释放操作就会造成内存泄漏，结果会造成堆中可供分配出去的内存数量越来越少。

9.4　小结

程序中的内容是通过段进行分类管理的。在传统的三个段中，.text 段用于存放处理器指令，.data 段用于存放初始化的全局和静态变量，.bss 段则用于存放初始化为 0 和未初始化的全局和静态变量。一个程序不管有多少个段，各段都属于程序、数据和调试信息三大类中的一种。

栈为函数参数和局部变量提供存储空间。局部变量所占用的内存空间是由编译器生成的指令自动分配与释放的，因此不存在像堆那样的内存泄漏问题。在多任务环境中，由于各任务的函数调用路径可以不同，所以每一个任务都有属于自己的栈空间。

堆中的内存在没有被分配出来之前，是整个系统所共享的。要从堆中获得内存，必须通过函数调用来实现。如果从堆中所分配获得的内存不再需要使用了，则必须调用相应的函数进行释放，否则会产生内存泄漏。

如果所需的临时内存空间比较小，则可以采用定义局部变量的方式以提高程序的执行效率；否则建议使用堆分配。



第 10 章 ABI/EABI规范，缔造程序兼容合约

ABI 是“Application Binary Interface”的缩写，即应用程序二进制接口，它是编译器和我们编写汇编代码时需遵循的规范。EABI 中的 E 是“Embedded”的首字母，它的提出是为了适应嵌入式系统资源更为有限这一现象。EABI 使得生成的程序更节约内存资源。

无论是哪一个厂商的编译器，它所生成程序文件的格式、函数调用时的栈帧结构都得符合 ABI 规范，否则就会出现用一个厂商的编译器生成的库无法被另一个厂商的编译器所用这种兼容性问题。

在 ABI 规范中所定义的内容包含（但不限于）以下内容。

- 软件安装。
- 底层系统信息。包括 C 语言数据类型字节占用大小的定义、结构和联合体的字节对齐处理方法、处理器寄存器的功能分配、栈帧结构、函数参数的传递和函数返回值的处理方法等。
- 目标文件格式。包括 ELF 文件的头、程序段、字符串表、符号表和程序的重定位。
- 程序的加载和动态链接。
- 库接口。包括共享库名、C 库、线程库、网络服务库、套接字库和系统数据接口。
- 开发环境。包括开发命令和软件打包工具。
- 应用程序可执行环境。

以上内容中我们将重点关注 ABI 中的底层系统信息这一部分，因为这一部分的内容对于嵌入式软件开发是最实用的，对于它的掌握有助于从处理器的角度去理解 C 语言。此外，对于其中栈帧内容的掌握，有助于我们理解函数调用时参数是怎样传递的、操作系统是如何实现多任务的等知识。

另外，ABI 只是一个总称，对于不同的处理器一定能找到与之相对应的 ABI 补充文档。在光盘的 embedded/doc/ABI&EABI 目录下，读者能找到 MIPS、PowerPC 和 x86 处理器的 ABI 补充文档。在嵌入式软件开发中，大部分情形下我们需要参考的是处理器对应的 ABI 补充文档。

10.1 定义基本数据类型

C 语言中存在各种基本数据类型，其所占用的字节数是由处理器对应的 ABI 规范定义的。图 10.1 列出了 x86 处理器 ABI 规范内所定义的各种类型所占内存的字节数，其中还定义了这些基本类型与 x86 处理器数据类型间的对应关系。

Type	C	Sizeof	Alignment (bytes)	Intel386 Architecture
Integral	unsigned char, signed char	1	1	signed byte
	unsigned char	1	1	unsigned byte
	unsigned short, signed short	2	2	signed halfword
	unsigned short	2	2	unsigned halfword
	signed int, unsigned int, signed long, unsigned long, enum	4	4	signed word
Pointer	*any-type* *, *any-type* (*) ()	4	4	unsigned word
Floating-point	float	4	4	single-precision (IEEE)
	double	8	4	double-precision (IEEE)
	long double	12	4	extended-precision (IEEE)

图 10.1

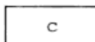
10.2 规范字节对齐处理

在 1.8 节谈到了处理器对于字节对齐的要求。对于一个结构体或联合体，如何对其中的成员变量或位域进行对齐也是由 ABI 规范定义的。图 10.2 是 x86 处理器 ABI 规范中所定义的结构体和联合体的变量对齐和填充方案，而图 10.3 则是针对位域的。

结构小于 4 个字节：

```
struct {
    char c;
};
```

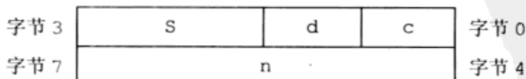
以单字节对齐，sizeof 的大小为 1



无字节填充：

```
struct {
    char c;
    char d;
```

以 4 字节对齐，sizeof 的大小为 8



```
    short s;
    long n;
};
```

内部填充：

```
struct {
    char c;
    short s;
};
```

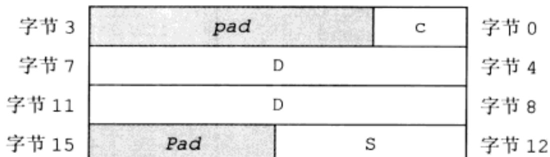
以 2 字对齐，sizeof 的大小为 4



内部和尾部填充：

```
struct {
    char c;
    double d;
    short s;
};
```

以 4 字节对齐，sizeof 的大小为 16



联合体内存布局：

```
union {
    char c;
    short s;
    int j;
};
```

以 4 字节对齐，sizeof 的大小为 4

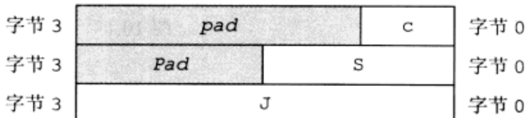
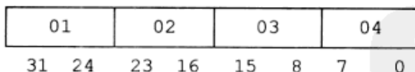


图 10.2

比特寻址：

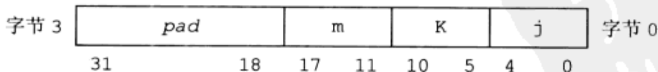
0x01020304



从右到左分配：

```
struct {
    int j:5;
    int k:6;
    int m:7;
};
```

以 4 字节对齐，sizeof 的大小为 4



边界对齐：

```
struct {
    short s:9;
    int j:9;
    char c;
    short t:9;
    short u:9;
    char d;
};
```

以 4 字对齐，sizeof 的大小为 12

字节 3	c	23*pad*18	17 j 9	8 s 0	字节 0
字节 7	15*pad*9	8 u 0	15*pad*9	8 t 0	字节 4
字节 11	*Pad*			D	字节 8

存储单元共享：

```
struct {
    char c;
    short s:8;
};
```

以 2 字节对齐，sizeof 的大小为 2

15 s 8	c

联合体内存布局：

```
union {
    char c;
    short s:8;
};
```

以 2 字节对齐，sizeof 的大小为 2

字节 1	*pad*	c	字节 0
字节 3	*pad*	7 s 0	字节 2

图 10.3

10.3　分配寄存器的功能

寄存器是处理器用来加工数据或运行程序的重要载体。有些处理器在设计时就规定好了部分寄存器的功能。比如，在 x86 处理器中，EIP 是指令寄存器，其指向处理器下一条要执行的指令在内存的位置；ESP 是栈指针寄存器；EBP 是栈帧（参见 10.4.1 节）基地址寄存器，等等。对于没有固定功能的寄存器，如果不加以规范就会带来兼容性问题。为了避免这种问题，ABI 规范中定义了这些寄存器的具体作用。图 10.4 列举了 x86 处理器 ABI 规范中定义的部分寄存器的功能。

寄存器名	功　能
EAX	用于存放函数的返回值
EDX	被除数寄存器，在进行除法运算时需要用到这个寄存器
ECX	计数寄存器，在进行移位或字符串操作时需要用到这个寄存器
EBX	局部变量寄存器
ESI	局部变量寄存器
EDI	局部变量寄存器

图 10.4

另外，很多的 RISC 处理器其寄存器更多，也需要通过 ABI 规范来定义每一个寄存器的作用，图 10.5 列出了 PowerPC 的 EABI 规范中所定义的每一个寄存器的作用。注意，其中的 R1 是用做栈指针寄存器的。

Register	Type	Used for
R0	Volatile	Language spedific
R1	Dedicated	Stack Pointer (SP)
R2	Dedicated	Read-only small data area anchor
R3 ~ R4	Volatile	Parameter passing / return values
R5 ~ R10	Volatile	parameter passing
R11 ~ R12	Volatile	
R13	Dedicated	Read-write small data area anchor
F0	Volatile	Language spedific
F1	Volatile	Parameter passing / return values
F2 ~ F8	Volatile	Parameter passing
F9 ~ F13	Volatile	
F14 ~ F31	Nonvolatile	

图 10.5

寄存器的功能定义好了以后，编译器在将 C 程序编译成汇编程序时就得遵守。比如，在 x86 处理器中，EBX、ESI 和 EDI 都被定义成函数的局部变量寄存器，当一个函数需要使用这些寄存器时，就需要先将这些寄存器的原值压入栈中先保存起来，因为上一个函数（即调用函数）可能也使用了这些寄存器，在函数返回时则从栈中恢复其原值。这些工作都是由编译器在幕后为我们做的。毫无疑问，当我们编写汇编程序时也得遵守规范，否则就会出现所编写的汇编程序无法与 C 程序协同工作的问题。

10.4 规定栈帧结构

从 C 编程语言的角度来看，程序功能的实现是通过一个个的函数调用来做到的，那栈在函数的调用过程中又是如何起作用的呢？接下来，让我们以 x86 为例加以讲解。

对于 x86 处理器，我们需要掌握两个与栈操作有关的寄存器：ESP（Extended Stack Pointer）和 EBP（Extended Base Pointer），其中 ESP 是指示当前的栈顶位置，而 EBP 是用于指示函数的栈帧基地址（后面简称基址）。

在 x86 的 ABI 规范中能找到图 10.6 所示的一张表，其示例说明了两个函数的栈帧布局，其中“Frame”列的“Previous”表示调用函数的栈帧结构，“Current”表示被调用函数的。这张表是站在被调用函数的角度观察而获得的。

该表还基于两个假设。第一，函数的返回值不是结构体和联合体。在 10.4.3 节中我们将看到当一个函数的返回值是结构体或联合体时，第一个参数并不是位于“8(%ebp)”处的，而是

“12(%ebp)”。第二，每个参数都是 4 字节大小的。在 10.4.1.3 节将就参数的传递和大小问题做进一步的探讨。从这张表中可以看出，函数的第一个参数位于“8(%ebp)”地址处，即以 EBP 为基地址、偏移量为+8 的内存空间（中的内容）。

Position	Contents	Frame	
0(%esp)	unspecified		High address
	...	Previous	
-4(%ebp)	variable size		
0(%ebp)	previous %ebp (optional)		
4(%ebp)	return address		
8(%ebp)	argument word 0	Current	
	...		
4n+8(%ebp)	argument word n		Low address

图 10.6

读者现在或许不能很好地理解该表，阅读本节中的后续小节有助于理解它。

10.4.1　栈帧的含义和作用

栈由栈帧组成，每个栈帧对应于一个（未执行完的）函数。接下来我们通过讲解栈帧的布局、形成和消亡来理解栈帧在函数调用时是如何起作用的。

10.4.1.1　栈帧的布局

图 10.7 所示是一个简单的测试程序，用于帮助我们了解栈帧。

```
00001: #include <stdio.h>
00002:
00003: //lint -e530 -e123
00004:
00005: void tail (int _param)
00006: {
00007:    int local = 0;
00008:    int reg_esp, reg_ebp;
00009:
00010: asm volatile(
00011:    // get EBP
00012:    "movl %%ebp, %0 \n"
00013:    // get ESP
00014:    "movl %%esp, %1 \n"
00015:    : "=r" (reg_ebp), "=r" (reg_esp)
00016:   );
00017:    printf ("tail (): EBP = %x\n", reg_ebp);
00018:    printf ("tail (): ESP = %x\n", reg_esp);
00019:    printf ("tail (): (EBP) = %x\n", *(int *)reg_ebp);
00020:    printf ("tail (): return address = %x\n", *(((int *)reg_ebp + 1)));
00021:    printf ("tail (): &local = %p\n", &local);
00022:    printf ("tail (): &reg_esp = %p\n", &reg_esp);
00023:    printf ("tail (): &reg_ebp = %p\n", &reg_ebp);
```

```
00024:     printf ("tail (): &_param = %p\n", &_param);
00025: }
00026:
00027: int middle (int _p0, int _p1, int _p2)
00028: {
00029:     int reg_esp, reg_ebp;
00030:
00031: asm volatile(
00032:     // get EBP
00033:     "movl %%ebp, %0 \n"
00034:     // get ESP
00035:     "movl %%esp, %1 \n"
00036:     : "=r" (reg_ebp), "=r" (reg_esp)
00037:    );
00038:     tail (_p0);
00039:     printf ("middle (): EBP = %x\n", reg_ebp);
00040:     printf ("middle (): ESP = %x\n", reg_esp);
00041:     printf ("middle (): (EBP) = %x\n", *(int *)reg_ebp);
00042:     printf ("middle (): return address = %x\n", *(((int *)reg_ebp + 1)));
00043:     printf ("middle (): &reg_esp = %p\n", &reg_esp);
00044:     printf ("middle (): &reg_ebp = %p\n", &reg_ebp);
00045:     printf ("middle (): &_p0 = %p\n", &_p0);
00046:     printf ("middle (): &_p1 = %p\n", &_p1);
00047:     printf ("middle (): &_p2 = %p\n", &_p2);
00048:     return 1;
00049: }
00050:
00051: int main ()
00052: {
00053:     int reg_esp, reg_ebp;
00054:     int local = middle (1, 2, 3);
00055:
00056: asm volatile(
00057:     // get EBP
00058:     "movl %%ebp, %0 \n"
00059:     // get ESP
00060:     "movl %%esp, %1 \n"
00061:     : "=r" (reg_ebp), "=r" (reg_esp)
00062:    );
00063:     printf ("main (): EBP = %x\n", reg_ebp);
00064:     printf ("main (): ESP = %x\n", reg_esp);
00065:     printf ("main (): (EBP) = %x\n", *(int *)reg_ebp);
00066:     printf ("main (): return address = %x\n", *(((int *)reg_ebp + 1)));
00067:     printf ("main (): &reg_esp = %p\n", &reg_esp);
00068:     printf ("main (): &reg_ebp = %p\n", &reg_ebp);
00069:     printf ("main (): &local = %p\n", &local);
00070:     return 0;
00071: }
```

图 10.7

这个小程序的每个函数中都嵌入了汇编代码，以便获得各函数运行时刻 ESP 和 EBP 寄存器的值。另外，每一个函数中都打印出了 EBP 寄存器所指向内存地址处的值，以及位于其后的函数返回地址，这样做的原因后面还会细讲。图 10.8 显示了这一程序的编译和运行结果。

```
$ make
$ ./release/stackframe.exe
tail (): EBP = 22cd08
tail (): ESP = 22ccf0
```

```
tail (): (EBP) = 22cd28
tail (): return address = 40120b
tail (): &local = 0x22cd04
tail (): &reg_esp = 0x22cd00
tail (): &reg_ebp = 0x22ccfc
tail (): &_param = 0x22cd10
middle (): EBP = 22cd28
middle (): ESP = 22cd10
middle (): (EBP) = 22cd58
middle (): return address = 401302
middle (): &reg_esp = 0x22cd24
middle (): &reg_ebp = 0x22cd20
middle (): &_p0 = 0x22cd30
middle (): &_p1 = 0x22cd34
middle (): &_p2 = 0x22cd38
main (): EBP = 22cd58
main (): ESP = 22cd30
main (): (EBP) = 22cd98
main (): return address = 61006e73
main (): &reg_esp = 0x22cd50
main (): &reg_ebp = 0x22cd4c
main (): &local = 0x22cd48
```

图 10.8

为了更好地理解输出结果中各数据间的关系，我们将其转化为图，如图 10.9 所示。图的左边还示例说明了栈的增长方向和栈的内存地址。黑色的箭头和寄存器名表示当前栈帧，否则用灰色表示。图中表示的是站在 tail()函数内所看到的栈布局，其中完整地示例说明了 tail()和 middle()两个函数的栈帧结构，以及 main()函数的一部分。

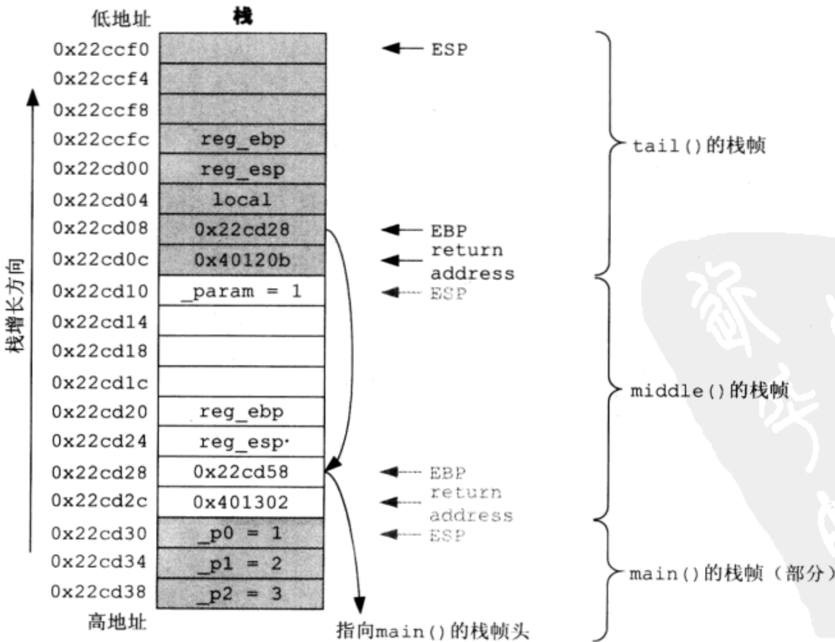


图 10.9

在通常情形下，每个函数都有自己的栈帧。各栈帧中存在一个域用于存放前一个调用函数的栈帧基址，通过这个域将所有调用与被调用函数的栈帧以链表的形式连在一起。栈帧的这种组织结构说明了为什么函数调用级数越多，所占用的栈空间也越大，也解释了为什么在嵌入式软件开发中我们需要小心使用递归函数。

10.4.1.2 栈帧的形成

为了方便讲解，我们还得获取图 10.7 所示的示例程序所对应的汇编代码片段，如图 10.10 所示。图中删除了 tail()函数汇编代码的中间部分，而只保留了头和尾用于创建和删除栈帧的内容。在汇编代码中，最左边列出了指令在内存中的地址，在接下来讲解栈帧中的返回地址（return address）信息时，其所指的内容就是指这一地址。

```
                    ~/embedded/build
$ objdump -d ./release/stackframe.exe > stackframe.txt
                    ~/embedded/build
$ vi stackframe.txt
00401130 <_tail>:
  401130:       55                              push   %ebp
  401131:       89 e5                           mov    %esp,%ebp
  401133:       83 ec 18                        sub    $0x18,%esp
  401136:       c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%ebp)
  40113d:       89 ea                           mov    %ebp,%edx
  40113f:       89 e0                           mov    %esp,%eax
  401141:       89 55 f4                        mov    %edx,-0xc(%ebp)
  401144:       89 54 24 04                     mov    %edx,0x4(%esp)
  401148:       89 45 f8                        mov    %eax,-0x8(%ebp)
  40114b:       c7 04 24 a0 20 40 00            movl   $0x4020a0,(%esp)
...... 显示结果有删减 ......
  4011e1:       c9                              leave
  4011e2:       c3                              ret

004011f0 <_middle>:
  4011f0:       55                              push   %ebp
  4011f1:       89 e5                           mov    %esp,%ebp
  4011f3:       83 ec 18                        sub    $0x18,%esp
  4011f6:       89 e8                           mov    %ebp,%eax
  4011f8:       89 e2                           mov    %esp,%edx
  4011fa:       89 45 f8                        mov    %eax,-0x8(%ebp)
  4011fd:       8b 45 08                        mov    0x8(%ebp),%eax
  401200:       89 55 fc                        mov    %edx,-0x4(%ebp)
  401203:       89 04 24                        mov    %eax,(%esp)
  401206:       e8 25 ff ff ff                  call   401130 <_tail>
  40120b:       8b 45 f8                        mov    -0x8(%ebp),%eax
  40120e:       c7 04 24 44 21 40 00            movl   $0x402144,(%esp)
  401215:       89 44 24 04                     mov    %eax,0x4(%esp)
  401219:       e8 da 01 00 00                  call   4013f8 <_printf>
...... 显示结果有删减 ......
```

图 10.10

现在假设程序运行在 main()刚调用 middle()函数的时刻，让我们看一看栈布局是如何发生变化的。程序一进入 middle()函数所运行的第一条指令位于内存地址 4011f0 处，在运行这一指令之前的栈结构如图 10.11 所示。此时的 EBP 还是指向 main()函数栈帧的头部，而 ESP 所指向

的内存中所存放的是程序返回到 main()函数的指令位置，后面分析 middle()函数对 tail()函数的调用时还将涉及这一点。

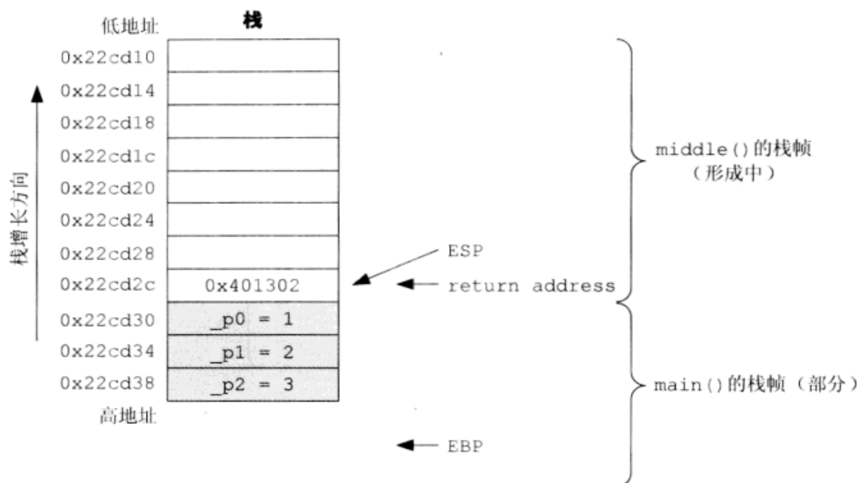


图 10.11

内存地址 4011f0～4011f3 的指令的作用就是形成 middle()函数的栈帧。第一条指令(位于内存地址 4011f0 处)是将调用函数（即 main()函数，middle()是被调用函数）的栈帧基址保存到栈上，这条指令是一个压栈操作。正是各函数内的这一操作，使得所有的栈帧连在了一起成为一条链。

第二条指令（位于内存地址 4011f1 处）将 ESP 寄存器的值赋值给 EBP 寄存器，也就是说，此时的 ESP 寄存器中保存的是 middle()函数的栈帧基址。请注意，基址并没有将用于保存返回地址的空间包含在内。

第三条指令（位于内存地址 4011f3 处）对 ESP 进行一个减操作，即将 ESP 向低地址处移动 24 个字节（对应于十六进制的 0x18)，移动 24 个字节的目地是为了在栈上腾出空间来存放局部变量和本函数需调用函数的传入参数。显然，函数内局部变量越大，则所减的数值就越大。

运行完了上面的三条指令以后，middle()函数的栈帧就形成了，如图 10.12 所示。图中还示例说明了 middle()函数内局部变量 reg_esp 和 reg_ebp 在栈帧中的位置。

位于内存地址 4011f6 和 4011f8 处的指令是我们在 middle()函数中所嵌入的汇编代码即用于获取此时 EBP 和 ESP 寄存器的值。4011fa 处的指令将 EBP 寄存器的值放入局部变量 reg_ebp 中，401200 处的指令将 ESP 寄存器的值放入局部变量 reg_esp 中。4011fd 和 401203 处的指令将 main()函数中传递过来的第一个变量_p0 的值拷贝到 ESP 寄存器所指向的内存中，为调用 tail()函数准备参数。此刻的栈空间如图 10.13 所示。

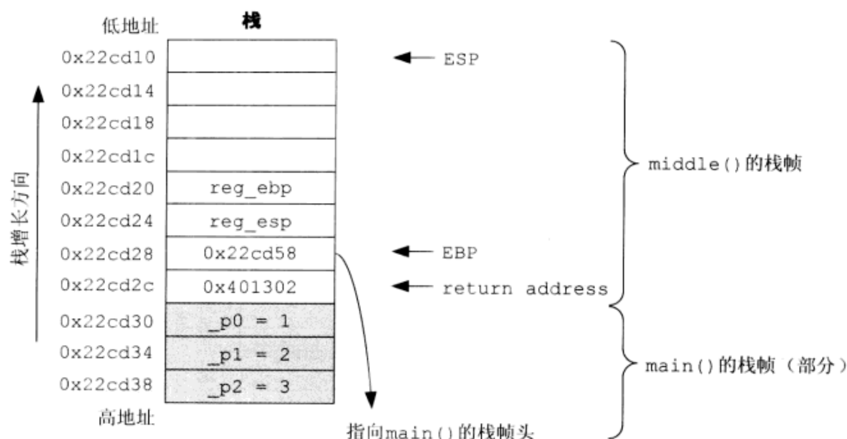


图 10.12

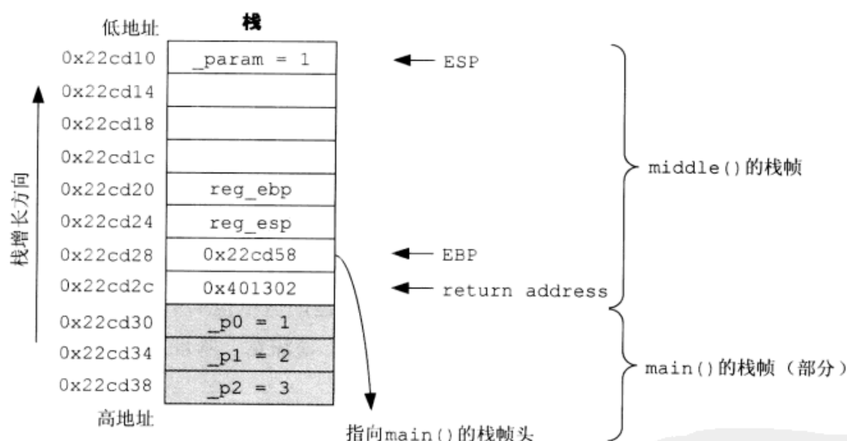


图 10.13

位于内存地址 401206 处的指令是调用 tail()函数的指令，这个调用会造成返回地址被压入到栈中，调用完了这条指令后的栈空间如图 10.14 所示。

所压入栈的返回地址是 40120b，从图 10.10 中可以看出这一地址指向的是 middle()函数内调用 tail()函数的后一条指令，也就是说，当 tail()函数返回时将从这一地址处继续运行程序。这条指令的调用也意味着进入了 tail()函数的栈帧，tail()函数也像 middle()函数那样采用相同的“手法”建立自己的栈帧。前面图 10.9 所示的内存布局，正是 tail()函数建立了栈帧时的。

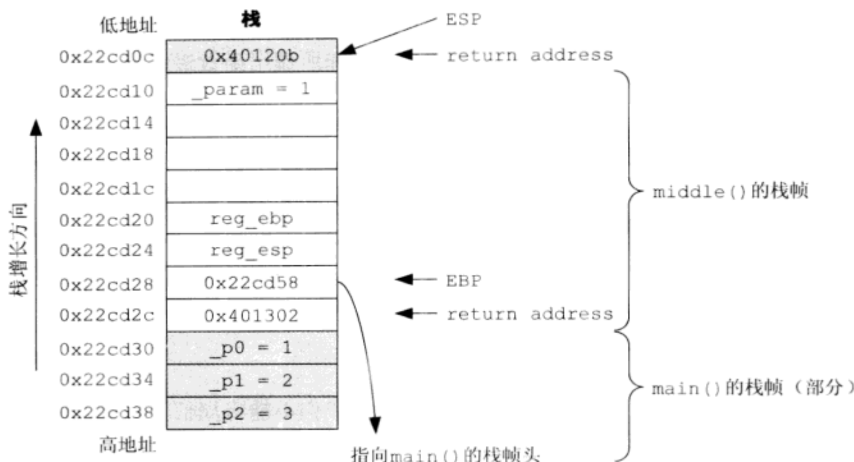


图 10.14

10.4.1.3 栈帧的消亡

下面让我们看一看在 tail()函数内进行函数返回时栈空间又是如何发生变化的。内存地址 4011e1 处的 leave 指令，其功能是将 ESP 寄存器的值设置为 EBP 寄存器的并做一次退栈操作，将退栈操作的内容放入 EBP 寄存器中。这条指令的功能等价于“mov %ebp, %esp; pop %ebp”，就是将 tail()函数所建立的栈帧去掉。这条指令执行完了后的栈布局与图 10.14 完全一样。tail()函数的最后是一条返回指令（位于内存地址 4011e2 处），用于将栈上（即 ESP 寄存器所指的位置）的内容弹出到 PC 寄存器中，其效果就是程序返回到了 middle()函数的 40120b 地址处。执行完这条指令后的栈结构与图 10.13 是一样的。

至此，我们完全了解了栈帧的形成与消亡。实际上，对于每一个 C 函数，编译器都会生成汇编代码在进入函数时创建其栈帧，以及从函数返回时将栈帧删除。在 x86 的 ABI 规范中，分别称这两部分为“前言”和“后序”，其大致代码分别如图 10.15 和图 10.16 所示。

```
prologue:
        pushl    ebp            // 保存上一函数的栈帧指针
        movel    esp,  ebp      // 设置本函数的栈帧指针
        subl    $80,  ebp       // 分配函数的栈帧空间（会因各函数的局变量大小不同而不同）
        pushl    edi            // 保存局部变量寄存器
        pushl    esi            // 保存局部变量寄存器
        pushl    ebx            // 保存局部变量寄存器
```

图 10.15

```
epilogue:
        popl    ebx             // 恢复局部变量寄存器
        popl    esi             // 恢复局部变量寄存器
        popl    edi             // 恢复局部变量寄存器
        leave                   // 恢复调用这一函数的栈帧指针
        ret                     // 返回到调用这一函数的函数内
```

图 10.16

在每一个函数的“前言”部分存在为栈帧分配大小的指令（比如图 10.15 中的“subl $80, %ebp”），C 编译器会根据函数中所存在的局部变量大小和所调用函数最多参数的个数来决定栈帧的大小。

另外在这两个图中分别存在对 EDI、ESI 和 EBX 的压栈及退栈操作。在 10.3 节中提到，EDI、ESI 和 EBX 是用做局部变量寄存器的。也就是说，如果这三个寄存器器在某函数（称之为函数 A）中使用了，而在其调用的函数（称之为函数 B）中也要用到它的话，那么函数 B 就必须在使用它们之前将它们保存起来，以便返回到函数 A 之前能恢复。但如果这两个函数都没有使用到这些寄存器，“聪明的”编译器会做出无须在“前言”中对其压栈的决定，以便提高程序的执行效率。

由于函数一旦返回其栈帧就不存在了，正因如此，我们不能将局部变量的指针作为函数的返回值。

如果读者现在回头看一看图 10.6 中的表，相信能更好地理解其含义。

10.4.2 函数参数的传递方法

在 ABI 规范中还定义了函数参数的传递方式和参数的压栈顺序。在 x86 处理器的 ABI 规范中规定，所有传递给被调用函数的参数都是通过栈来完成的，其压栈的顺序是以函数参数从右到左的顺序。在图 10.7 的第 54 行，当 main()函数调用 middle()函数时所传入 3 个参数在栈中的布局可以从图 10.9 的下方找到，参数压栈的顺序是_p2、_p1 和_p0。

在 x86 处理器上，当向一个函数传递参数时，所有的参数最后形成的是一个数组。由于采用从右到左的压栈操作，所以数组中参数的顺序（比如，从下标 0 到下标 2）与函数从左到右的顺序是一致的（_p0、_p1 和_p2）。因此，在一个函数中如果知道了第一个参数的地址和各参数占用字节的大小，就可以通过访问数组的方式去访问每一个参数。

10.4.2.1 整型和指针参数的传递

整型参数的传递在前面已经看到了，而指针参数的传递与整型是一样的。这是因为，在 32 位 x86 处理器上整型的大小与指针的大小都是一样的，都占 4 个字节。来自 x86 处理器 ABI 规范中的图 10.17 总结了这两种类型的参数在栈帧中的位置关系。注意，该表是基于 tail()函数的栈帧而言的。

Call	Argument	Stack Address
tail (1, 2, 3, (void *)0);	1	8(%ebp)
	2	12(%ebp)
	3	16(%ebp)
	(void *)0	20(%ebp)

图 10.17

10.4.2.2　浮点参数的传递

浮点参数的传递与整型其实是相似的，唯一的区别就是参数的大小。在 x86 处理器中，浮点类型占 8 个字节，因此在栈中也需要占用 8 个字节。来自 x86 处理器 ABI 规范的图 10.18 示例说明了浮点参数在栈帧中的位置关系。图中，调用 tail()函数的第一个和第三个参数都是浮点类型，因此各需要占用 8 个字节，三个参数共需要占用 20 个字节。图中的 word 类型的大小是 4 个字节。

Call	Argument	Stack Address
tail (1.414, 1, 2.998e10);	word 0, 1.414	8(%ebp)
	word 1, 1.414	12(%ebp)
	1	16(%ebp)
	word 0, 2.998e10	20(%ebp)
	word 1, 2.998e10	24(%ebp)

图 10.18

10.4.2.3　结构体和联合体参数的传递

结构体（struct）和联合体（union）参数的传递与前面提到的整型、浮点参数相似，只是其占用字节的大小需视数据结构的定义不同而异。但是无论如何，结构体在栈上所占用的字节数一定是 4 的倍数。这是因为在 32 位的 x86 处理器上栈宽是 4 字节的，因此编译器也会“很聪明地”对结构体进行适当的填充以使得结构体的大小满足 4 字节对齐的要求。

上面讲解的内容都是以 x86 处理器为例的。对于一些 RISC 处理器，比如 PowerPC，其参数传递并不是全部通过栈来实现的。从图 10.5 中 PowerPC 处理器寄存器的功能分配表可以看出，R3～R10 共 8 个寄存器用于整型或指针参数的传递，F1～F8 共 8 个寄存器用于浮点参数的传递。当所需传递的参数个数小于 8 时，根本不需要用到栈。

图 10.19 是一个在 PowerPC 处理器上多参数传递的例子，图 10.20 则是处理器寄存器的分配和栈帧在参数传递时的布局。

```
typedef struct {
   int int1_, int2_;
   double double_;
} parameter_t;

void middle ()
{
   parameter_t p1, p2;
   int int1, int2, int3, int4, int5, int6;
   long double long_double;
   double double1, double2, double3, double4, double5, double6, double7, double8, double9;

   tail (int1, double1, int2, double2, int3, double3, int4, double4, int5, double5,
      int6, long_double, double6, double7, p1, double8, p2, double9);
}
```

图 10.19

General Purpose Registers	Floating Point Registers	Stack Frame Offset
R3: int1	F1: double1	8(%ebp): pointer to p2
R4: int2	F2: double2	12(%ebp): (padding)
R5: int3	F3: double3	16(%ebp): word 0 of double9
R6: int4	F4: double4	20(%ebp): word 1 of double9
R7: int5	F5: double5	
R8: int6	F6: double6	
R9: pointer to long_double	F7: double7	
R10: pointer to p1	F8: double8	

图 10.20

可以看出，结构体和 long double 参数的传递是通过指针来完成的，这一点与 x86 处理器完全不同。在 PowerPC 的 ABI 规范中规定，对于结构体的传递仍采用指针的方式，而不是像 x86 处理器那样将结构从一个函数的栈帧中拷贝到另一个函数的栈帧中，显然 x86 处理器的方式更低效。由此看来，“在实现函数时，其参数应当尽量用指针而不是用结构体以便提高效率”这一原则对于 PowerPC 处理器上的程序并不成立。但无论如何，养成函数参数传指针而非结构体这一编程习惯还是有益的。

10.4.3 函数返回值的返回方法

在 x86 处理器上，当被调用函数需要返回结果给调用函数时存在两种情形。一种是返回的数据是标量（比如整型、指针等），在这种情形下，返回值将会放入 EAX 寄存器中。如果返回的是浮点数，则返回值是放在协处理器的寄存器栈上的。

另一种情形是函数需要返回结构体或联合体（非标量）。这种情形需要通过栈来完成。为了了解在这种情形下栈帧的作用，我们可以借助图 10.21 所示的示例程序。

```
00001: #include <stdio.h>
00002:
00003: //lint -e530 -e123 -e428
00004:
00005: typedef struct {
00006:     int i0_;
00007:     int i1_;
00008:     int i2_;
00009: } func_return_t;
00010:
00011: func_return_t foo (int _param)
00012: {
00013:     func_return_t local;
00014:     int reg_esp, reg_ebp;
00015:
00016: asm volatile(
00017:     // get EBP
```

```
00018:     "movl %%ebp, %0 \n"
00019:     // get ESP
00020:     "movl %%esp, %1 \n"
00021:     : "=r" (reg_ebp), "=r" (reg_esp)
00022: );
00023:     printf ("foo (): EBP = %x\n", reg_ebp);
00024:     printf ("foo (): ESP = %x\n", reg_esp);
00025:     printf ("foo (): (EBP) = %x\n", *(int *)reg_ebp);
00026:     printf ("foo (): return address = %x\n", *(((int *)reg_ebp + 1)));
00027:     local.i0_ = 1;
00028:     local.i1_ = 2;
00029:     local.i2_ = 3;
00030:     printf ("foo (): &_param = %p\n", &_param);
00031:     printf ("foo (): return value = %x\n", *(((int *)&_param) - 1));
00032:     printf ("foo (): &local = %p\n", &local);
00033:     printf ("foo (): &reg_esp = %p\n", &reg_esp);
00034:     printf ("foo (): &reg_ebp = %p\n", &reg_ebp);
00035:     return local;
00036: }
00037:
00038: int main ()
00039: {
00040:     int reg_esp, reg_ebp;
00041:     func_return_t local = foo (100);
00042:
00043: asm volatile(
00044:     // get EBP
00045:     "movl %%ebp, %0 \n"
00046:     // get ESP
00047:     "movl %%esp, %1 \n"
00048:     : "=r" (reg_ebp), "=r" (reg_esp)
00049: );
00050:     printf ("main (): EBP = %x\n", reg_ebp);
00051:     printf ("main (): ESP = %x\n", reg_esp);
00052:     printf ("main (): &local = %p\n", &local);
00053:     printf ("main (): &reg_esp = %p\n", &reg_esp);
00054:     printf ("main (): &reg_ebp = %p\n", &reg_ebp);
00055:     return 0;
00056: }
```

图 10.21

在这个示例程序中，main()和 foo()函数内都定义了一个类型为 func_return_t 的 local 变量，且 foo()的返回值类型也是 func_return_t。毫无疑问，两个 local 变量的内存都将分配在各自函数的栈帧中，那 foo()函数的 local 变量的值是如何通过函数返回值传递到 main()函数的 local 变量中的呢？编译这个程序并运行以观察其结果，如图 10.22 所示。图 10.23 示例说明了在 foo()函数内所看到的栈布局。

```
                    /embedded/build
$ make
                    /embedded/build
$ ./release/return.exe
foo (): EBP = 22cd18
foo (): ESP = 22cce0
foo (): (EBP) = 22cd58
```

```
foo (): return address = 40125a
foo (): & param = 0x22cd24
foo (): return value = 22cd3c
foo (): &local = 0x22cd00
foo (): &reg esp = 0x22cd10
foo (): &reg ebp = 0x22cd0c
main (): EBP = 22cd58
main (): ESP = 22cd20
main (): &local = 0x22cd3c
main (): &reg esp = 0x22cd4c
main (): &reg ebp = 0x22cd48
```

图 10.22

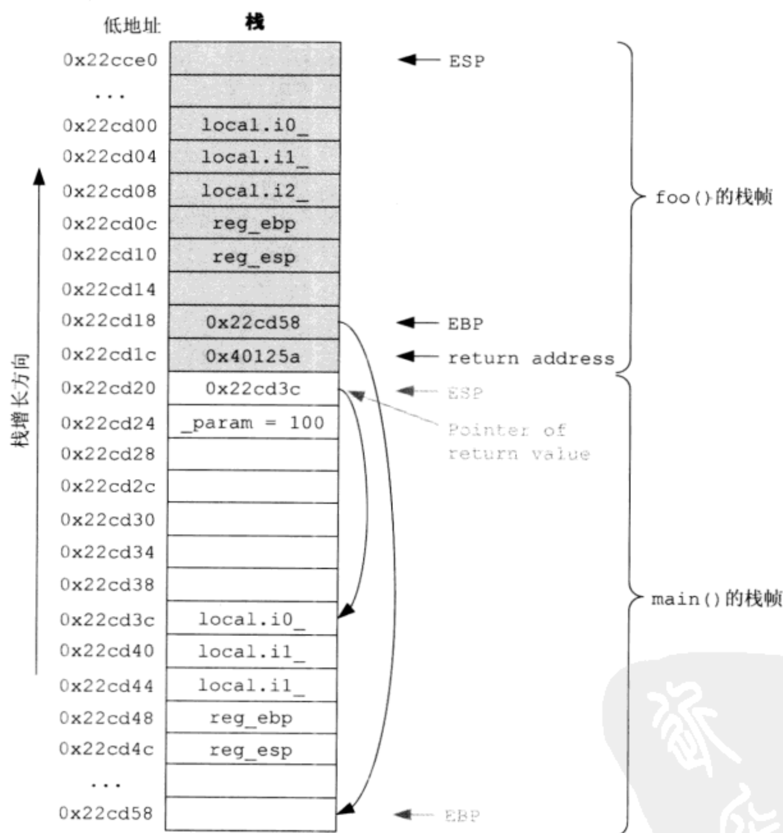


图 10.23

从图中可以看出，main()函数调用 foo()函数时除了将 foo()函数所需的参数压入到栈中外，还将局部变量 local 的地址也压入到栈中，当 foo()函数在进行函数返回时会将它的 local 变量的值通过这一指针拷贝到 main()函数的 local 变量中。正是因为存在这一拷贝操作，所以在 x86 处理器上将结构当做函数返回类型是相对耗时的。

从上面的分析我们还注意到一种现象，当函数是以结构体或联合体作为返回值时，函数的第一个参数是存放在 12（%ebp）位置处的，因为中间多了一个返回值的地址。

10.5　小结

本章通过分析 x86 处理器上的程序，结合对应的 ABI 规范深入地介绍了结构体和联合体的对齐及填充方式、栈帧的含义和作用、函数参数的传递方法、函数返回值的返回方法，等等。虽然分析是针对 x86 处理器的，但是，其机理对于所有的处理器都是相同的。

深入地理解 ABI 规范有助于更加深入地理解 C 语言，也有助于我们从处理器的角度理解编译器的幕后行为。

练习与思考

1. 通过栈帧的形成原理，分析栈溢出是如何发生的。

2. 在 x86 处理器上，写一个小程序以打印出某一函数被调用时的调用栈。

3. 如果被调用函数需要 2 个参数，而我们在调用时却传递了 3 个参数，在这种情形下对于被调用函数(注意，只针对被调用函数)的功能有影响吗？反过来，如果被调用函数需要 3 个参数但我们只传递了 2 个参数，又会如何呢？



第 11 章

混淆指针与数组所导致的问题

给程序“治病”可以说是工程师的日常工作。不少问题通过检查验证能很容易地明白问题所在，但是有些问题的成因却让人难以理解。本章介绍的就是 C 语言中很容易犯但不容易想明白的一个问题。

在 C 语言中，一个数组变量可以理解成定义了一个指向数组的指针，但这只是一种理解。在使用时还得将其与指针区分开，否则将会导致错误。下面通过分析混淆它们所导致的问题为例，揭示 C 语言中一个鲜为人知的特点。

11.1 问题示例

图 11.1 示例说明了正确和错误引用定义在 define.c 文件中的数组变量的例子程序。

```
char g_name [] = {'Y', 'u', 'n', '\0'};
```

```
#include <stdio.h>

extern char g_name [];

int main ()
{
    printf ("%c\n", g_name [0]);
    return 0;
}
```

```
#include <stdio.h>

extern char *g_name;

int main ()
{
    printf ("%c\n", g_name [0]);
    return 0;
}
```

图 11.1

define.c 定义了一个数组变量 g_name。correct.c 则是一个包含了对 g_name 变量进行正确声

明的文件，它将 g_name 声明成一个外部定义的字符数组。相反，在 wrong.c 中错误地将 g_name 变量声明成一个外部定义的字符指针。两个测试程序都试图打印出 g_name 变量中的第一个字符。图 11.2 所示是编译并运行两个可执行程序 correct.exe 和 wrong.exe 的操作结果。

```
$ gcc -g define.c correct.c -o correct.exe

$ ./correct.exe
Y

$ gcc -g define.c wrong.c -o wrong.exe

$ ./wrong.exe
     5 [main] wrong 3060  cygtls::handle exceptions: Error while dumping state (probably
corrupted stack)
Segmentation fault (core dumped)
```

图 11.2

结果显示，当数组被错误地声明成指针时，程序虽然能正常地编译出来，但运行结果却不对且导致了崩溃。

11.2　问题分析

要探究出错的根源，我们需要回顾一下 C 语言中的指针和数组的内存模型。后面的分析都是基于程序是运行在 32 位 x86 处理器上的。

11.2.1　数组的内存模型

了解内存模型最好的方法不是查看 C 语言的相关参考书，作者比较喜欢通过编写小程序的方式加深理解。图 11.3 中的示例程序有助于我们观察数组的内存模型，其编译和运行结果如图 11.4 所示。

```
00001: #include <stdio.h>
00002:
00003: int main ()
00004: {
00005:    char name [] = {'Y', 'u', 'n', '\0'};
00006:
00007:    printf ("     Addr of name: %p\n", &name);
00008:    printf ("  Addr of name[0]: %p\n", &name[0]);
00009:    printf ("Content of name[0]: 0x%x (%c)\n", name[0], name[0]);
00010:    printf ("  Addr of name[1]: %p\n", &name[1]);
00011:    printf ("Content of name[1]: 0x%x (%c)\n", name[1], name[1]);
00012:    printf ("  Addr of name[2]: %p\n", &name[2]);
00013:    printf ("Content of name[2]: 0x%x (%c)\n", name[2], name[2]);
00014:    printf ("  Addr of name[3]: %p\n", &name[3]);
00015:    printf ("Content of name[3]: 0x%x (%c)\n", name[3], name[3]);
00016:    return 0;
00017: }
```

图 11.3

```
$ gcc arraymodel.c -o arraymodel.exe

$ ./ptrmodel.exe
      Addr of name: 0x22ccd0
   Addr of name[0]: 0x22ccd0
Content of name[0]: 0x59 (Y)
   Addr of name[1]: 0x22ccd1
Content of name[1]: 0x75 (u)
   Addr of name[2]: 0x22ccd2
Content of name[2]: 0x6e (n)
   Addr of name[3]: 0x22ccd3
Content of name[3]: 0x0 ( )
```

图 11.4

图 11.5 所示是通过运行结果所获得的 name 数组的内存模型。从图中可以看出，C 语言中数组变量的名可以认为代表了数组的开始地址，从数组的开始地址处每一个数组元素是按程序中出现的顺序依次排列的。

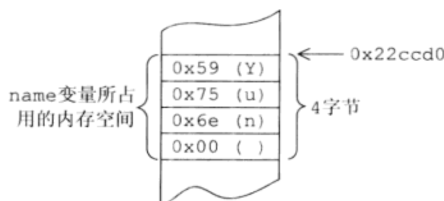


图 11.5

11.2.2　指针的内存模型

图 11.6 是一个用于帮助我们理解指针内存模块的示例程序，其编译和运行结果如图 11.7 所示。

```
00001: #include <stdio.h>
00002:
00003: int main ()
00004: {
00005:     char name[] = {'Y', 'u', 'n', '\0'};
00006:     char *p_name = &name[0];
00007:
00008:     printf (" Addr of name: %p\n", &name);
00009:     printf (" Addr of p_name: %p\n", &p_name);
00010:     printf (" Content of p_name: 0x%x\n", p_name);
00011:     printf ("Content of p_name[0]: 0x%x (%c)\n", p_name[0], p_name[0]);
00012:     printf ("Content of p_name[1]: 0x%x (%c)\n", p_name[1], p_name[1]);
00013:     printf ("Content of p_name[2]: 0x%x (%c)\n", p_name[2],p_name[2]);
00014:     printf ("Content of p_name[3]: 0x%x (%c)\n", p_name[3], p_name[3]);
00015:     return 0;
00016: }
```

图 11.6

```
$ gcc ptrmodel.c -o ptrmodel.exe

$ ./ptrmodel.exe
       Addr of name: 0x22ccd0
     Addr of p_name: 0x22cccc
  Content of p_name: 0x22ccd0
Content of p_name[0]: 0x59 (Y)
Content of p_name[1]: 0x75 (u)
Content of p_name[2]: 0x6e (n)
Content of p_name[3]: 0x0 ( )
```

图 11.7

图 11.6 中的第 5 和 6 行分别定义了两个变量——name 和 p_name，上一节我们已经知道了 name 变量的内存模型，这里主要关注 p_name 指针变量的内存模型。图 11.8 所示是通过运行结果所获得的两个变量在内存中的布局示意图。

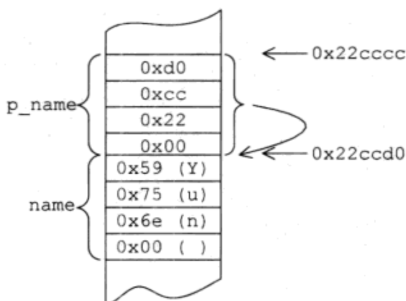


图 11.8

从图中可以看出，p_name 变量占用了 4 个字节的内存空间，它指向的是 name 数组变量的开始地址。当采用 p_name[0]这种格式来引用所指向的数组时，将返回 name 数组中第一个元素的内容，这从输出结果可以看出来。

11.3　问题成因

现在，我们已经了解了数组和指针的内存模型，是时候分析一开始的那个问题是如何发生的了。图 11.9 示例说明了出现问题时程序的相关变量在内存中的布局。在这个图中对于 g_name 变量可以从两个角度来看，一是从 define.c 文件的角度看（图的左边），二是从 wrong.c 的角度看（图的右边）。

当从 define.c 的角度看时，g_name 是定义在这一文件中的一个字符数组。由于 g_name 是一个初始化好的全局变量，所以它的内存是分配在.data 段的，这可以从图 11.10 所示的运行命令的结果中看出。

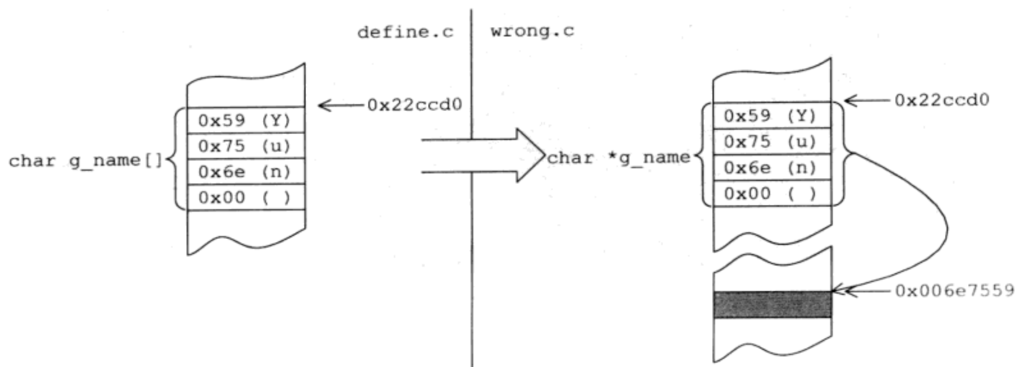


图 11.9

```
$ gcc -c define.c

$ nm -n define.o
00000000 b .bss
00000000 d .data
00000000 t .text
00000000 D _g_name

$ objdump -s -j .data define.o

define.o:     file format pe-i386

Contents of section .data:
 0000 59756e00                             Yun.
```

图 11.10

但从 wrong.c 的角度看时，g_name 是一个在其他文件中定义的一个指针变量，并且它只是引用这一变量。当 define.o 和 wrong.o 被链接在一起时，两个文件中的 g_name 变量将被当做同一个。由于在 wrong.c 文件中 g_name 是被声明成一个外部定义的字符指针变量，所以在引用 g_name[0]进行打印时，它将使用前面 11.2.2 节中所介绍的指针内存模型进行访问。如此一来，程序将“Yun”字符串的值理解为指针值，而试图读取 0x006e7559 地址处的一个字节，这个字节表示为图 11.9 中的灰色区块。由于这个地址对于 wrong.exe 进程并不是有效的地址，所以程序就会产生一个“Segmentation fault”错误。这是导致问题的首要原因。

从 4.2 节中的内容可知，在链接器最终生成 wrong.exe 程序之前，define.c 和 wrong.c 会分别被编译生成 define.o 和 wrong.o 文件。而每一个目标文件中记录了文件内部所定义和依赖外部的符号。从图 11.11 使用 nm 查看 wrong.o 目标文件的符号信息可以看出，g_name 变量（即 _g_name）并没有在文件中定义，前面的“U”就是表示“Undefined”的。当一个文件被编译生成目标文件后，目标文件中将不包含任何的 C 语法信息。记住这一点非常重要！

```
$ nm -n wrong.o
```

```
         U _g_name
         U _printf
00000000 b .bss
00000000 d .data
00000000 r .rdata
00000000 t .text
00000000 T _main
```

图 11.11

链接的过程就是将所有的同名符号合成一个。当 define.o 和 wrong.o 被链接器链接生成可执行程序时，链接器并不知道 define.o 文件中定义的 g_name 符号在 C 语言中的类型与 wrong.o 文件中所需要的并不相同，因此将它们进行配对。由此看来，导致错误的进一步原因是，链接器发现不了目标文件中的符号不匹配问题。

11.4　预防措施

如果链接器不能发现符号不匹配问题，那只能指望编译器了。在 wrong.c 文件中将原本是数组的 g_name 变量声明成指针，这一声明 define.c 并不知道。也就是说，对于 g_name 这个变量的类型信息存在两个“信息孤岛”，一个来自 define.c 文件，另一个来自 wrong.c 文件，且这两个“信息孤岛”之间没有任何的联系。我们可以通过为“信息孤岛”建立联系来预防这类问题。

在编写程序时，应当将一个外部需要引用的变量或者函数放在一个头文件中，然后让定义和引用它们的源文件同时包含它，这种包含就为各“信息孤岛”间建立起了联系。有了这种联系，编译器就能帮助我们发现问题。

图 11.12 是采用这一方法得到的程序文件，其中 define.h 是新增的，我们将 g_name 的声明放到了其中，并且在 define.c 和 wrong.c 文件中同时包含它。

```
extern char* g_name;
```

```
#include "define.h"

char g_name [] = {'Y', 'u', 'n', '\0'};
```

```
#include <stdio.h>
#include "define.h"

int main ()
{
printf ("%c\n", g_name [0]);
return 0;
}
```

图 11.12

图 11.13 所示是对更改后的文件进行编译的结果。从结果可以看到，编译器指出了 g_name 存在不同的定义和声明，这正是我们所希望看到的。

```
$ gcc define1.c wrong1.c -o wrong1.exe
define1.c:3: error: conflicting types for 'g_name'
define1.h:1: error: previous declaration of 'g_name' was here
```

图 11.13

关于定义和声明不匹配的问题，同样也会发生在函数上。在项目中之所以会出现直接采用 extern 进行外部变量或函数的引用，大部分情况就是为了图省事，看来图省事却可能带来问题。因此，我们在规划和管理一个项目时，要考虑整个项目中的头文件组织得可以被方便地包含，并且将“永远采用头文件作为定义和引用的桥梁”作为一个编程好习惯。

11.5 小结

通过分析混淆指针与数组所产生的问题，我们分别回顾了数组和指针的内存模型，并揭示了链接器并不关心 C 语言的语法这一事实。

为了避免出现混淆指针与数组所产生的问题，我们需要养成总是将头文件作为（变量和函数的）定义和引用的桥梁这一编程好习惯。



第 12 章

volatile，让我保持原样

在 1.4 节中指出，对于内存映射的 I/O 端口我们可以像内存读写那样与外设进行交互。图 12.1 是一个用于激活某一外设的函数。

```
#define DEVICE_READY     0x01

void device_activate (int *_port)
{
    *_port = DEVICE_READY;
    while (*_port != DEVICE_READY);
}
```

图 12.1

device_activate()函数的_port 输入参数是外设的控制端口地址，通过向该寄存器的比特 0 写 1 的方式来激活它，外设准备好了以后，控制端口的比特 0 将被外设置为 1。device_activate()函数正是通过不断地查询该位的状态来判断外设是否初始化好了。请注意，外设的寄存器并不像内存那样，我们写了 1 进去读出来的也一定是 1，这完全取决于外设的行为。

device_activate()函数的功能在不使用编译优化选项时是正常的，这可以从图 12.2 的反汇编程序看出。但是，当使用编译优化选项时它的功能就不正常了，反汇编结果如图 12.3 所示。

```
$ gcc -g -c device.c

$ objdump -S -d device.o

device.o:     file format pe-i386

Disassembly of section .text:

00000000 <_device_activate>:
#define DEVICE_READY    0x01

void device_activate (int *_reg)
{
   0:   55                          push   %ebp
   1:   89 e5                       mov    %esp,%ebp
    *_reg = DEVICE_READY;
   3:   8b 45 08                    mov    0x8(%ebp),%eax
   6:   c7 00 01 00 00 00           movl   $0x1,(%eax)
```

```
  while (*_reg != DEVICE_READY);
   c:   8b 45 08                    mov    0x8(%ebp),%eax
   f:   8b 00                       mov    (%eax),%eax
  11:   83 f8 01                    cmp    $0x1,%eax
  14:   75 f6                       jne    c <_device_activate+0xc>
}
  16:   5d                          pop    %ebp
  17:   c3                          ret
```

图 12.2

```
$ gcc -O2 -c device.c

$ objdump -S -d device.o

device.o:     file format pe-i386

Disassembly of section .text:

00000000 <_device_activate>:
   0:   55                          push   %ebp
   1:   89 e5                       mov    %esp,%ebp
   3:   8b 45 08                    mov    0x8(%ebp),%eax
   6:   c7 00 01 00 00 00           movl   $0x1,(%eax)
   c:   5d                          pop    %ebp
   d:   c3                          ret
```

图 12.3

从图 12.3 中可以看出，在编译器使用优化选项的情形下，device_activate()函数中的 while 语句会被优化掉，也就是 device_activate()函数设置完了激活位就返回，而没有确认设备是否真的被激活了。这是因为编译器“聪明地”认为：对寄存器的 0 比特设置了 1 以后读入的值也一定是 1，所以那个 while 语句是多余的。

造成这种结果是因为编译器将_port 参数所指向的地址当做内存来处理而不是端口。我们得告诉编译器那其实是端口，这需要依靠 volatile 关键字。使用 volatile 关键字更改后的代码如图 12.4 所示。

```
#define DEVICE_READY    0x01

void device_activate (volatile int *_port)
{
    *_port = DEVICE_READY;
    while (*_port != DEVICE_READY);
}
```

图 12.4

图 12.5 所示是再一次使用优化选项编译和反汇编的结果。从图中可以看出，这次编译器没有优化掉 while 语句。

```
$ gcc -O2 -c device.c
```

```
$ objdump -S -d device.o

device.o:     file format pe-i386

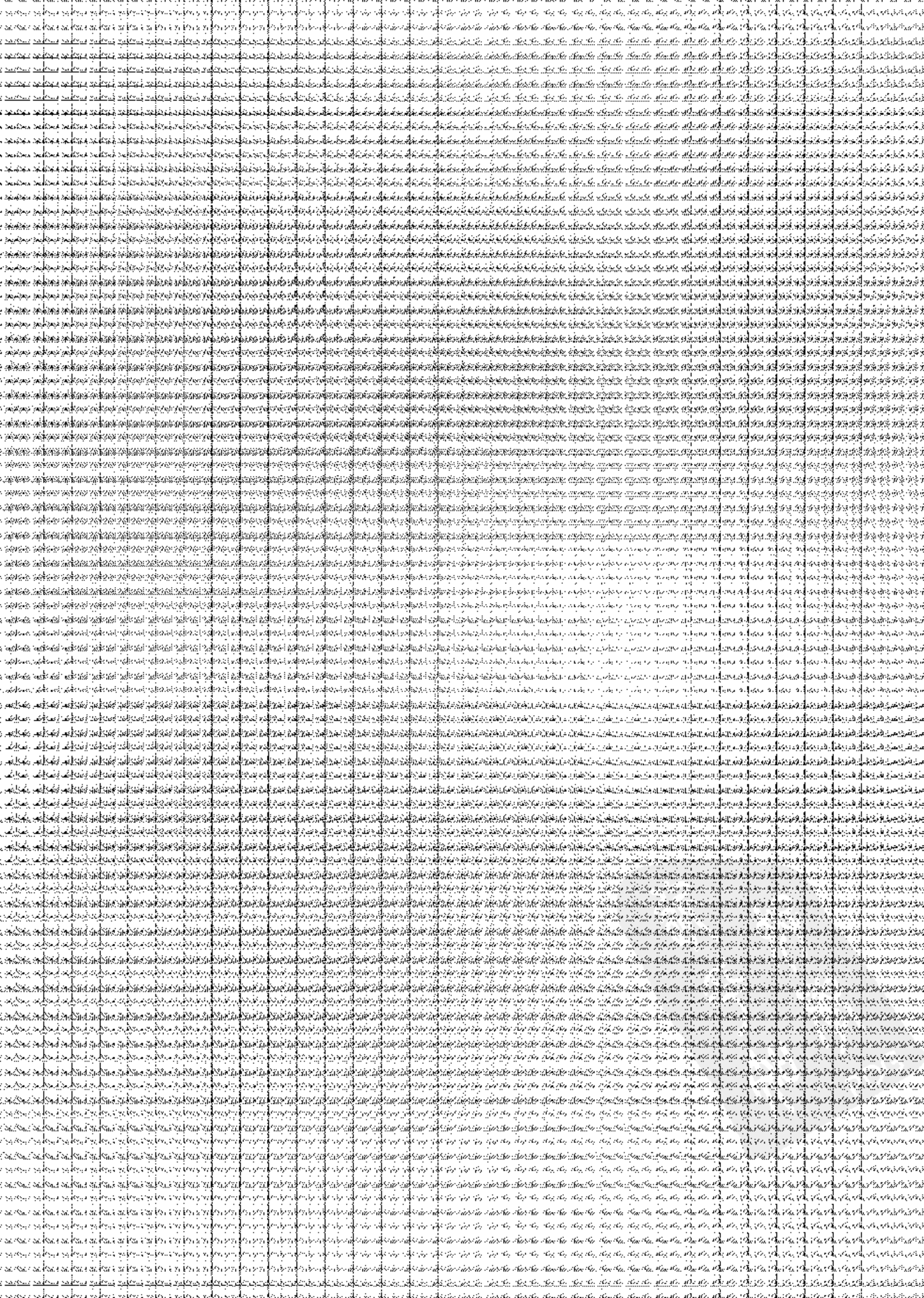
Disassembly of section .text:

00000000 < device activate>:
   0:   55                          push   ebp
   1:   89 e5                       mov    esp, ebp
   3:   8b 55 08                    mov    0x8( ebp), edx
   6:   c7 02 01 00 00 00           movl   $0x1,( edx)
   c:   8d 74 26 00                 lea    0x0( esi, eiz,1), esi
  10:   8b 02                       mov    ( edx), eax
  12:   83 e8 01                    sub    $0x1, eax
  15:   75 f9                       jne    10 < device activate+0x10>
  17:   5d                          pop    ebp
  18:   c3                          ret
```

图 12.5

编写与外设打交道的程序时需要注意运用 volatile 关键字，否则可能出现程序在不采用编译优化选项时能正常运行，而一旦使用编译优化选项就不正常这种“怪”现象。





设计篇

高质量的软件一定源于良好的软件架构，良好的软件架构源于设计。说到软件设计，工程师很容易想到数据结构的定义，但出色的软件设计应当以塑造概念为主。设计质量在软件行业似乎并没有得到应有的重视，或许是因为具备出色设计能力的工程师稀缺造就了这种现状。好的设计是软件产品的质量之本。在第 13 章解释了什么是软件设计、为什么设计是产品质量之本，也指出在现实工作中几种阻碍改善设计的常见观念，并就如何提高设计能力提出了一些建议。另外，还给出了几个放之四海皆适用的设计原则。

模块化设计方法早已深入人心，因为通过模块化这种“分而治之”的方法能有效地降低设计的复杂度。当一个系统比较复杂时，模块间不可避免地会产生复杂的依赖关系，且有可能出现“牵一发而动全身”这种状况。第 14 章介绍的分层与分级的概念为模块的管理提供了一种参照系，所引入的模块管理实现也被运用到了本书任何一个有需要的角落。

无论软件的功能如何，软件在运行期间总是会出现错误的。说到软件的错误管理，相信不少人会不寒而栗。软件开发如果只需考虑一切正常的情形，那么开发活动绝对是既省时又省力，但这不现实！出色的软件产品，一定存在一套有效的错误管理机制。第 15 就错误管理进行了探讨。

软件开发无小事！除了将我们的精力集中于程序实现外，还应关注像项目目录结构管理这样的“小事”。好的项目目录结构具有书架功能、意识引导功能和加速新成员上手的功能。在第 16 章我们将共同探讨项目目录结构的功能和出色目录结构的特点。

如果每一个项目都是从头开始做起，而不是基于已有的实现进行再开发或定制，那么要让每个产品都实现高质量会是一件遥不可及的事。无论是软件企业或是个人，都应当致力于打造可复用的软件模块，以便积累知识、经验乃至教训。在第 17 章我们将看一看为什么要通过平台与框架开发这种方式来构建企业和个人的核心竞争力。

软件开发效率的获得除了采用复用已有代码的方法外，还得考虑通过软件设计去缓解开发设备不足、设备使用效率不高这类问题，这需要运用第 18 章所提出的可开发性设计思想。可开发性设计思想是对平台与框架开发技术的一种运用扩展。

除了以上这些关于设计的话题外，在其他的篇章中还嵌入了实时性设计（24.6.2 节）和可测试性设计（28.7 节）。之所以不将它们放入本篇，是为了让本书更具可读性。

第 13 章 设计，软件质量之本

需求错误如果直到开发阶段的后期才发现，修正它将额外花费好几倍的努力(成本)，这在软件行业已是一种共识。另外，需求错误由于它的外部表现性，容易被用户和公司管理层所感知。正因如此，需求分析的质量总是能引起大家的重视。但是让开发团队深陷痛苦泥沼的，往往并非是由需求引起的，而是由不良设计导致的。然而设计质量却经常被忽视。

设计质量被忽视的一个重要原因是，它不像需求那样具有外部表现性。只要软件产品在功能上满足需求，即使设计质量的不尽人意达到了“掀了马桶盖却导致楼塌了”[①]之势也可能得不到足够的重视，因为“它还能工作”。

不良设计的影响实在是大，它使得软件产品的维护和扩展举步为艰、难以测试和查错，从而直接影响项目的开发效率和产品的最终质量，以及工程师的生活质量。

13.1 软件设计是什么

定义软件设计并非易事，因为每个人都有不同的理解。作者给出的定义是：软件设计是一系列创造活动，是借助编程语言以简单和优雅的方式表达并解决现实需求的一门科学和艺术。

首先，软件设计是一门科学。科学的特点是有规律可循，因此软件设计者需要掌握相关的专业知识。比如需要学习数据结构、计算机组成原理、编程语言等内容，这是专业性的体现。科学知识通常更容易被量化和评估。拿数据结构为例，一种算法比另一种算法是否更优，可以从其时间冗余度和空间冗余度进行衡量。

其次，它还是一门艺术。设计的最终产物并不只是代码，还包含了设计者在创造这个软件世界的过程中的分析、抽象、取舍等。一个好的设计必然给人带来美感，也让人值得欣赏。

表达方式的“简单”是指程序的实现直截了当，易于理解。简单性是通过深邃的思考后去除“旁枝末叶”而获得的，它是设计人员良好洞察力的体现。在 13.6.2 节就简单性的衡量标准

---

① 从好友于善成那第一次听到“掀了马桶盖却导致楼塌了”这句话时，感同身受。

做了进一步的解释。“优雅”的表达方式强调软件架构的合理性，设计之美也正是集中体现在软件架构上的。

软件设计是通过一系列的创造活动来完成的。首先，软件设计是一个不断提炼和抽象的过程。设计者在设计之初会觉得有很多因素需要考虑，随着设计的深入，将从众多的因素中提炼出关键因素并忽略其他因素。软件设计也是一个不断抽象的过程，需要从众多的表象中找到它们的共性，并从共性入手进行表达。

比如，现在假设有两组数据，分别是 A 组的{1,2,3,4,5}和 B 组的{4,7,12,19,28}，且两组数据中的元素间存在位置上的映射关系，即通过 A 组的 1 将获得 B 组的 4、2 将获得 7…。实现映射功能的函数最直接的做法是在函数体中通过 switch 语句进行判定并返回。但我们只要稍微分析一下，就可以找到它们的共性：$B = A^2 + 3$。这就是一个抽象过程。

软件设计是一个塑造模型(或概念)的过程。模型除了帮助实现需求外，还展示了“模样”和“行为模式”。塑造模型的好处是便于设计者与其他人进行沟通，通过这种方式所获得的数据结构和函数也更容易被理解和使用，甚至让其具有生命性。塑造模型时应当注意概念的完整性和实现的一致性。上段例子中的 $B = A^2 + 3$ 便是我们通过分析归纳得到的一个数学模型。

软件设计是一个取舍的过程。软件实现的多样性决定了软件的设计过程存在大量的“选择题”。所以设计过程是痛苦的，除非设计主题很简单。

软件设计是一个分而治之的过程。随着发展，软件的复杂度和规模都在急骤地增长，我们只有通过分而治之这种降低局部复杂度和规模的方式，才有可能更好地做好设计工作。分而治之也是我们常说的模块化设计方法。

软件设计是一个在有限理性范围内追求完美的过程。出色的设计是在设计者不断追求完美的过程中获得的，但追求完美需要注意度。由于大多数项目存在时间和资金预算上的压力，因此我们需要在一定的约束条件下追求完美。

13.2　软件质量的概念

一说到软件质量，很容易想到软件缺陷。因此，缺陷少潜移默化地成为了高质量软件的代名词。但这种认识是片面的。

从软件用户的角度来看，缺陷越少说明质量越高这应当是合理的。但是，从开发团队的角度来看，如果一个软件的缺陷的确少，但开发团队为了实现这一目标却要花费并不相称的超额努力，这样的软件就不能称之为高质量。这两个角度的软件质量标准，作者分别称为用户级和开发团队级。

之所以将软件质量分成两个级别，是因为设计质量的外部不可见性。如果我们只从用户级

别来考察软件质量，就会忽视设计质量的重要性，也就只局限在“只要功能正常就是好软件”这一认识范围中。设计质量可能在用户级无法反映出来，但在开发团队级一定能被正确反映。

真正高质量的软件不仅需要满足用户级的软件质量，还需要满足开发团队级的软件质量。一个满足开发团队级质量的软件，能使开发团队保持良好士气，并从容、更有效率地从事软件开发工作。

以实现开发团队级的高质量为目标才可能从长远的角度保证用户级的高质量，否则，用户级的高质量有可能只是昙花一现。

缺陷数量是业内不少质量管理体系的评估指标。这种评估方法存在局限性，因为只停留在用户级，而没有切中“高质量”的要害。这类质量体系很难促使产品质量获得本质的提高。

要做到开发团队级的高质量，我们必须保证软件的设计质量。作者想通过图 13.1 来指出设计质量与产品质量的关系。从图中不难看出，项目规模越大，则设计质量对产品质量就越重要。

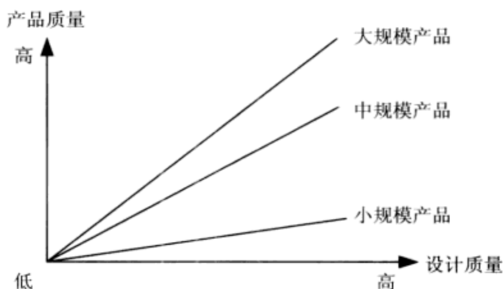


图 13.1

主导设计一旦确立了以后，产品的质量水准就基本确定了。尽管验证阶段对于产品质量很重要，但验证工作做得再好也不能从本质上改变软件产品的质量水准。这也是为什么有些项目在投入市场以后，在不更改主导设计的情形下，无论修复多少缺陷仍一团糟的原因。

那如何保证设计质量呢？这很容易让人想到通过开发流程和认证，比如敏捷软件开发方法、CMMI 认证等。其实认证的本质还是流程。但是，高设计质量是不能简单地通过流程而获得的，因为流程所控制的是有形的因素，而软件设计过程很多内容是无形的。由于软件设计中艺术成份的非直观性，造成设计质量不易被量化以便加以评估。

要保证设计质量，人是关键——具备良好设计能力的人。他们理解什么是软件设计，以及拥有自己的设计思想并积极付之于项目实践。由于掌握软件设计比理解业务需求更难，因此具备设计能力的人在行业中很缺乏。现实中，开发团队很少因为理解不了业务需求而不能开发产品，但因为不理解软件设计而深受煎熬却并不少见。

软件设计能力很抽象，但不能因为抽象就不重视它，软件开发团队应当以培养和打造自己

的核心设计骨干作为自己的重要内容之一。由于具备软件设计能力的工程师是稀缺资源，不能指望项目团队中的每一个人都是真正的设计者，也就是说，软件开发工程师并不等同于软件设计师，这一点我们必须有清楚的认识。

13.3　阻碍改善设计的常见观念

既然软件设计如此重要，那么忽视它就是一种战略短视行为。软件工程师最重要的工作内容理应是进行真正有创造性的软件设计工作，而不应当只忙于简单地“修补漏洞”。漏洞是得补，但得补得有艺术、有深度，而不是采用头痛治头、脚痛治脚这样的方式。没有深度的修补方式注定是在为将来埋下更大的定时炸弹，也可以预见未来的软件维护工作将愈加困难。

软件开发很容易进入混乱状态，要从中走出必须采用逐步改善设计的方式，而不是等待“颠覆性时刻”的到来。之所以出现不少项目安于现状，即使受尽煎熬也无意通过改变走出困境，是因为存在几种常见的错误观念。

13.3.1　测试是替罪羊或救命稻草

测试是软件质量保证的重要一环，是验证软件功能是否与需求相一致的必需方法，但是千万不能将其当做软件质量的唯一保证方法，更不应当让测试成了软件缺陷的“替罪羊”或质量保证的“救命稻草”。

现实工作中存在这样一种现象，只要软件发现了新的缺陷，就有人会指责测试部门“为什么没有测出这个问题？”理论上，测试部门应当对最终的产品质量负责，因为它们是质量卫士，但实际上要做到这一点并不容易。原因在于测试部门无论如何努力工作，也不可能构造出所有的测试用例(test case)，对于大型系统和分布式系统更是如此。这也是为什么软件行业存在“测试只能证明失败”这种观点的原因。

提出“别让测试成了替罪羊”这种观点的目的不在于为测试工程师进行责任开脱，而在于提醒软件开发工程师不要忘记从设计的角度去审视“这种缺陷能否通过设计避免？”，或者思考“是不是现在的设计注定了会出现这么多的缺陷，是否可以通过改善设计来真正有效、彻底地解决问题？”

软件开发工程师应当明白，如果软件设计没有做好，测试工程师也很难单方面保证最终的软件质量。注意：他们也只能保证用户级的软件质量。最终的软件质量一定来源于开发工程师所做出的优良设计，以及测试工程师别出心裁的测试用例设计。

设计没有做好却将测试当做“救命稻草”是一件很可怕的事，不光项目最终做不好，参与其中的每一个人都将背负沉重的包袱。一个根基不好的建筑，再怎样通过外部加固也不能成为优质工程。而软件质量的根基即是设计质量。千万不要将质量保证的口号变成“测试，测试，再测试！”，而应是“设计，设计，再测试！”。

最后再指出一点：对于一个设计很糟糕的软件，开发工程师可以试着问“如果我是测试工程师，能否通过设计出完善的测试用例去保障软件的最终质量呢？”。如果自己也觉得不行，那说明只能通过改善设计去尝试着改变这一问题的答案。

13.3.2 资源永远不足

很多项目的困境是由不良设计所造成的。当项目处于困境中时，不能一味地指望投入更多的资源。“资源不足”在很多情形下是我们不愿意改变现状和承担风险的一个借口。

当我们身处困境时，不能幻想有一天上司说“接下来的半年我们不再开发新的功能，而是致力于改善设计以提高产品质量”。即使听到这句话，那很有可能是指“因为我们的软件质量太差了，用户都不愿意用了，只能等质量好一点他们才考虑使用”。在这种“非做好不可”情况下再考虑改善设计，团队的压力就更大了。我们必须选择更好的途径——“乱中求治”。

“乱中求治”是指在现有资源的配置下，长期坚持“抠”出一点资源来逐步改善现有设计，以期最终实现设计质量质变的方式让项目走出困境。

“乱中求治”的方法是为了避免重写软件这种颠覆性时刻的到来，既能有效地控制风险，也可以给项目团队更大的余地。

我们不能乐观地认为重写软件就一定能做出更好的设计，设计做不好的瓶颈可能在于团队的能力，而能力在短期内是无法获得突破性提升的。“乱中求治”能为项目团队提供一种持续锻炼能力的机会。如何在困境中找到不良设计的根源并通过改进加以解决是很具挑战性的工作内容，每克服一个困难，项目团队的能力就获得了一定的提升，而且这类提升将随着长期坚持而产生放大效应。

除了软件设计质量的提高外，“乱中求治”还可以帮助打造团队的文化，一种积极面对困境的创造性文化。资源不足不应成为阻碍改善设计的永远借口！

13.3.3 不改变就可以规避风险

另一种阻碍项目团队进行设计改进的思想来源于风险控制意识，具有这种意识的人担心因为改变反而增加了风险。

风险与创新似乎是矛盾的，很多组织大力提倡创新但却严格控制风险。严格控制风险意味着很难获得改变的机会，那创新也很有可能被扼杀。风险应当有大小之分，如果严格控制所有的风险，那就是间接否认风险有大小之别。控制风险固然重要，但应当有个度，对于小风险的事情应当倡导团队去尝试。

处于混乱状况的项目，其风险不会因为不去改变而消失，运用复杂度守恒定律去理解的话，现在不去改变那将意味着将来会有（更大的）风险。现在不做改变可能短期内不会暴露已存在

的风险，但从长远来看暴露是必然的。不选择承担短期风险的原因可能是“过了今年后不知这个项目还做不做”，或者是“管它呢，过了今年再说，到时也不关我的事了”。

过度的风险控制意识大多来源于项目管理者。就作者的工作经验来看，大部分的软件工程师都勇于承担一定的风险，因为这使工作变得有趣且能学到更多的东西，甚至有工程师为了学习新东西而不顾风险的存在。作为管理者控制风险是对的，只是要注意方法和度。

让团队处于一定可控风险的压力之下对于团队的发展是有益的。别忘了除了控制风险外，管理者很重要的一个责任是培养团队。一点风险都没有的工作一定很无趣，也无法激发团队的工作激情和创造力。团队如果不给工程师一点更具风险性的工作，那很难让工程师得到成长，也很难将他们长期留在团队中。控制风险的一种有效方法就是运用前面提到的“乱中求治”思想。

管理者担心改变的风险很有可能是对团队能力的不信任，或者团队的能力真的让人不信任。但团队的能力从何而来？能力往往是通过改变和犯错积累的，就这个角度来说，在可控的前提下管理者应当放手让团队去做一些小小的风险性尝试。因为不信任团队而害怕改变所带来的风险，或许是在为自己制造一个悖论。

13.4　如何提高设计能力

首先，需要对软件设计有精神上的追求，并不断追求设计的完美性。梦想之所以有可能成为现实，是因为我们会去“想”并为之付出努力，软件设计能力的提高也不例外，设计能力并不会因为不去追求而“不小心”获得且水平很高。具备软件设计追求的人，会在设计的第一时间积极思考，以试图找到更优方案，也会随着产品的演变而反思是否存在更好的设计，更会在需要时接受改善设计这一挑战帮助团队走出困境。

提高设计能力的另一个途径是实践加模仿。知识和方法只有对之熟悉了以后才能运用自如，要熟悉它就得通过实践，且一开始是模仿性的实践，设计能力的提高也是如此。一开始可以看别人的设计，通过看懂它、掌握它让其在我们的大脑中留下印象，以后碰到类似设计主题时，我们就会想起它并依样画葫芦地做。类似的模仿多了我们就能领悟其中的精髓并把握各种设计方法的本质，乃至最后自己也能创造性地思考出更优的设计。

实践和模仿的目的是为了最终形成自己的设计思想，设计思想的形成需要通过思考去做到。设计思想是设计时所遵守的各条原则，是设计原则的集合。有的设计一看就觉得好，是因为它符合某些设计原则。而思考的目的就是从各种好的设计中，找出隐藏在背后的原则。

从一个好的设计中找出隐藏在背后的设计原则需要我们具有良好的洞察力。作者在高中时购买过一本武术书——《截拳道》，这本书讲解了李小龙创立的截拳道。截拳道更加注重搏击效率，因此它的招式都是以实用、直接打击对手为目的。诚然，在格斗的过程中，没有人会优

先考虑自己的招式是如何的“酷”，否则就是找打，而应注重如何在格斗过程中占上风乃至最后取胜。李小龙以他的洞察力从纷繁复杂的招式中发现最为简练的那些，通过对招式的简化来提高搏击效率。

洞察力不只对于找出设计背后所隐藏的设计原则有用，其实在整个软件开发乃至人生中都有着十分重要的作用。良好的洞察力有助于发现表象背后的本质，寻找出问题的根源。洞察力的获得，需要我们养成严密思考的习惯，这种习惯也会体现在谈吐、文档等之上。作者相信，一个思考不严密的人不可能做出优良的软件设计。

设计能力的提高意味着将掌握更多的设计原则，能力的提高过程也是对设计原则进行精化的过程。理论上，应尽可能让各设计原则所涵盖的内容是正交的。设计不是简单地运用每一个原则，还需运用原则时有很好的平衡感。

追求设计之美是提升设计能力的原动力，实践和模仿起到的作用是熟悉各种“零星”的好设计，而思考则是帮助领悟各种“零星”的好设计并找出隐藏在其背后的设计原则，进而形成自己完整的设计思想。

13.5 设计模式、设计原则和设计思想

设计模式在面向对象领域有着极为深远的影响，它已成为面向对象开发工程师的基本术语之一。与设计原则相比，设计模式更具体，因此也更具指导性。当做一个设计时，可以参照各种设计模式以找出可能适合的一种或几种。与设计模式不同的是，设计原则却更加抽象，也更难驾驭。

设计思想则更抽象，它之所以有效也正是因为其抽象性。现实的软件项目纷繁复杂，如果某一方法过于具体，则很难将这一方法从一个项目借用到另一个项目；反之，如果更具抽象性，则更容易在多个项目中对之加以运用，但实施起来更难“落地”。图 13.2 示例说明了设计思想、设计原则和设计模式之间的关系。

用树来做个比方的话，设计思想是树干，设计原则是树枝，而设计模式则是树叶。要了解一棵树长什么样，不能只看树叶。树叶虽然是第一个出现在我们眼前的事物，但它过于具体，也隐藏了树枝的模样。树枝则因为被树叶所遮挡，因而更“抽象”，但其表达能力却更强，通过树枝的样子，我们完全可以推测出树叶长出来时树的大致模样，树干则比树枝更具“抽象性”。

设计思想的形成是一个螺旋上升的过程，是一个不断从具体到抽象的过程。从设计模式到设计原则，以及最后形成自己的设计思想，就是一种在抽象层次上不断提高的过程。提高自己的设计能力，应以打造自己的设计思想作为最终目标。

设计模式并不是本书的探讨内容，读者可以参考 Erich Gamma、Richard Helm、Ralph Johnson 和 John M. Vlissides 合著的《设计模式》加深理解。

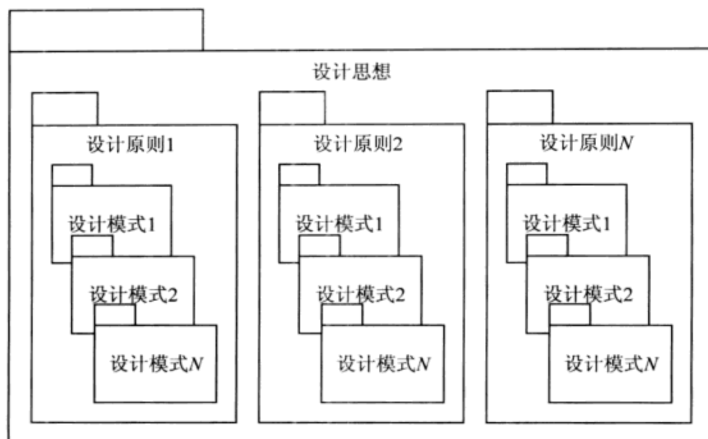


图 13.2

13.6　放之四海皆适用的设计原则

一种设计适合某一项目但未必适用于另一个项目，要掌握每一个项目所适用的各种设计是不现实的。经验告诉我们，通过使用规则有助于让我们的大脑掌控更多的东西。同样地，借助设计原则将使得我们能更好地掌控软件设计。一旦掌握一定的设计原则，无论做怎样的项目，通过运用它们将有助于做出(更)好的设计。

由于设计原则带有一定的抽象性，因此它在设计中所起到的作用仍是指导性的。同一个设计原则在不同的项目中所起的效果也可能不同，恰当地运用各原则需要不断地思考和练习。有了枪就一定能打到猎物吗？没有枪法的练习，枪只是枪而不是真正的狩猎工具。

下面将介绍作者在工作中常使用的部分设计原则，并通过提供设计实例帮助读者理解。由于在创作本书期间，对一致性原则和完整性原则没有碰到合适的素材，所以没有将它们写入本书。在以后碰到合适的素材时，作者将通过博客与大家分享。

13.6.1　以人为本

在 13.1 节阐述什么是软件设计时指出，软件设计是一个塑造模型的过程。塑造模型时，应以现实世界为参照，这就是以人为本设计原则的具体含义。

设计出来的软件除了达到它所应满足的需求外，另一个很重要的功能是它能向他人传达设计意图。一个设计不可能永远由一个人去维护，这就存在设计者与后继者的沟通问题。沟通的形式有多种，比如写文档、口授等，但最有效的沟通方式是让设计自身具有“自说明”的能力。

我们在现实生活中积累了大量的生活经验，这些经验为沟通带来了极大的效率。比如，一说到刀大家就能立即明白刀是什么，而不需要解释为“是一种由铁做成的用于切割东西的工

具”。既然生活经验对于沟通效率是如此的重要，那设计时在软件世界所创造的模型就应当迎合人们的生活经验。如果在软件世界里将刀设计成一种用于拧螺丝的工具，那多少显得有点怪异，因为这与我们的生活常识完全不符。

《数据结构》这类书籍所传授的内容主要立足于软件世界，它们是软件世界的基石，对于软件开发工程师之间的沟通也很有效，但却不够生动，表达力也不够。设计应当尽可能在软件世界中反映现实世界，只有这样才会更生动，更容易让人理解。通过以人为本设计原则所设计的软件能让涉众知其一后根据自己的经验推测出二，这显然将大大地提高被设计软件的“自说明”能力。

下面以一个在 VxWorks 操作系统上采用内存管理单元保护任务内存池的设计为例，来帮助读者进一步理解“以人为本”这一设计原则。

假设在一个运行 VxWorks 操作系统的嵌入式设备上存在多个任务，且所有的任务是以共享系统内存的方式运行的，也就是说，即使是任务 A 所专有的内存池任务 B 也可以因为出错而意外访问。通过采用内存管理单元(参见 1.10 节)实现内存池保护功能，将有助于防止程序中因为未初始化指针等因素所造成的非法内存池访问问题。在探讨设计方案之前，需要先了解设计用例(use case)。

图 13.3 示例说明了三个任务和三个内存池，在每一个内存池中也标识出它可以被哪个(些)任务读写。当任务 A 处于活动状态时(注意：每一时刻只能有一个任务是处于活动状态的)，内存池 1 和 2 可被读写，而内存池 3 不可以；同理，当任务 B 处于活动状态时，内存池 2 和 3 将变为可被读写但内存池 1 不可以；最后，当任务 C 处于活动状态时，三个内存池都不可被读写。

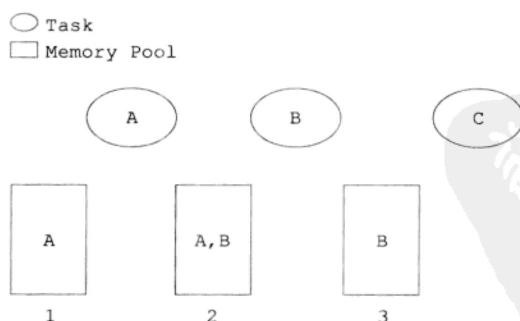


图 13.3

根据“以人为本”设计原则，我们需要在生活中找到能表达这一设计用例的相似场景，并将之引入到软件世界中。很多公司都有员工牌且员工牌与公司的门禁系统是关联在一起的，每个人能进入公司哪些办公区或实验室都是通过员工牌来进行权限管理的。如果将员工牌的这种现实模型搬到软件世界中就能让最终的设计结果具备良好的“自说明”能力。

图 13.4 是引入了员工牌这一概念后的示意图，其中的 mbadge 是“memory badge”的简写。“badge”有“徽章”的意思，也可用于表示员工牌[②]。

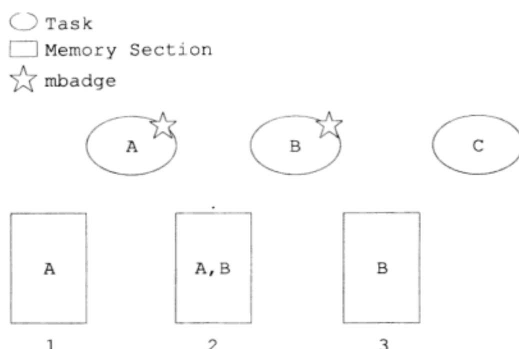


图 13.4

引入 mbadge 以后，很自然会问这样一个问题：任务对于每一个内存池的访问权限是记录在哪儿呢？相信读者会想到应当是记录在 mbadge 数据结构中，这一点与现实世界稍微有点不同[③]，但仍不失直观性。由于任务 C 不需要对三个内存池进行访问，所以它不需要 mbadge。

有了 mbadge 的概念以后，就很容易向他人解释。如果某一任务要使用被内存管理单元保护的内存池，第一步需要做的是让任务拥有一个 mbadge，接着对 mbadge 设置该任务对各内存池的访问权限。很明显，这样的解释也易于被他人理解。

“以人为本”这一设计原则在于强调在软件世界中塑造与现实相似的模型，或者说创造概念。当有了模型以后，后面怎么具体实现就有了很好的方向性，因为现实世界就是一个参照，数据结构和函数的命名也将随之确定。模型的存在能让最后的代码读起来像有生命似的，而不只是一堆死板的数据结构和函数。

这一原则还带给我们一些启发。在编写设计文档时，应该先描述设计所展现的模型是什么，而不应一上来就讲数据结构，这更有助于文档的读者掌握设计思想。在进行设计审查时，也应当问设计者“设计表达的模型是什么”，这有助于了解设计质量。大多不良设计正是因为没有模型而让人觉得混乱。

某种程度上，“以人为本”设计原则的运用与后面将要谈到的“通过命名传达设计意图”设计原则是分不开的。

② 作者创作这本书时是摩托罗拉杭州研发中心的员工，由于公司是以“badge”来称呼员工牌的，所以在做软件设计时就想到了采用这一称呼。

③ 在现实世界中，员工牌的权限是存储在中央数据库中的，而不是在员工牌内。员工牌所起到的只是像名字那样的标识作用。

13.6.2 追求简单性

设计的简单性与灵活性似乎如影随行，要分清它们有时并不容易。让我们从一个简单的例子开始了解追求简单性这一设计原则。

假设我们需要编写一个函数，它可以实现“home.example.net!{sm=1}user@other.example.net”这样格式的字符串中花括号(含)部分的内容。图 13.5 示例说明了一种实现。

```
void decoration_remove (char _user_name[])
{
   char realm [NAME_MAX];
   char user [NAME_MAX];

   sscanf (_user_name, "%[^{]", realm);
   sscanf (_user_name, "%*[^}]%s", user);
   sprintf (_user_name, "%s%s", realm, &user[1]);
   return 0;
}
```

图 13.5[④]

在该实现中，通过调用 sscanf()函数并使用正则表达式分离出花括号前后的字符串，然后再通过 sprintf()函数将花括号的前后字符串拼接在一起放回传入参数_user_name 中。尽管这一实现能达到目的，但作者在审查这个函数实现时指出它过于复杂，不过有人却主张这样的实现具有更好的灵活性。

在作者看来，设计的简单性不只体现在代码少上，还应包含概念易于理解、所涉及的知识点尽可能少、程序执行效率更高等。上面的设计在代码行数上是简单的，但它使用了没有必要的正则表达式知识点，且两个 sscanf()函数的调用需要更多的处理器时间。况且使用正则表达式的实现其实也没有带来灵活性，假如将来输入字符串的格式改变了，同样无法避免修改函数实现的问题。图 13.6 示例说明了另一种实现。

```
void decoration_remove (char _user_name[])
{
char *p_char = _user_name, *p_from, *p_to;

   while (*p_char != 0) {
      if (*p_char == '{') {
         p_from = p_char;
      }
      else if (*p_char == '}') {
         p_to = p_char ++;
         goto found;
      }
      p_char ++;
```

④ 这里假设函数的输入参数一定是符合格式的，所以函数没有对之做有效性检查。

```
    }
    return;

found:
    while (*p_to ++ != 0) {
        *p_from ++ = *p_to;
    }
}
```

图 13.6

这一实现，从可理解性及程序性能上都优于前一个实现，且没有使用正则表达式知识点。

我们目前只讨论了函数实现方面的简单性与灵活性，这与软件设计原则似乎存在很大的差距。但别忘了，软件设计其实最终在函数的实现上有所体现。在软件的设计过程中，当我们面临简单性与灵活性的取舍时，我们如何做出选择呢？

如果某一灵活性设计直接带来了简单性，那灵活性和简单性间就没有矛盾，自然也就没有选择问题。但如果灵活性与简单性之间存在矛盾时，我们可以通过回答以下两个问题来帮助做选择。

- 灵活性设计的依据是否来源于现有需求？灵活性不能全以自我感觉为依据，而是需要先找到支撑灵活性的需求。如果需求不存在，那么说明所谓的灵活性极有可能毫无意义，或许还会为将来带来更大的不确定性。不少采纳灵活性的设计者声称“如果将来变成了……的话，这种设计将更加的灵活”，出现这种陈述，往往意味着现有需求并没有要求这样的灵活性。在这种情形下，如果“那个如果”根本就不可能发生，那意味着我们并不需要所谓的灵活性。但是，当“那个如果”成立时，我们还得问第二个问题。
- 灵活性设计的采纳与否对于现在和将来的工作量是否会产生大的差异？如果现在和将来的工作量并不会因为灵活性设计的运用而产生大的差异，那我们应当放弃灵活性。

具有简单性的设计除了更容易被理解和维护外，还意味着我们不致于过度设计。过度设计是冠冕堂皇的浪费，也往往会给我们带来更大的复杂度。

一个出色的设计，除非设计者已经具备了相同或相似的经验，否则很难一步到位。请不要相信每一次选择灵活性设计将使我们最终构建出一个出色的软件架构；相反，高质量的设计是在追求简单性的过程中，在需求出现的情形下通过逐步的灵活性演进而获得的。

灵活性设计在不少情形下是一个优美的陷阱，要避免踏入这一陷阱，需要我们抵挡住它的诱惑，方法就是通过问自己前面的两个问题。另外，我们不应为了“显摆”而追求没有必要的灵活性，用直截了当的方式解决问题并不等同于“做事不专业”。

13.6.3 让模块善始善终

在软件行业，模块化设计早已深入人心，因为通过模块化这种“分而治之”的方法能有效地降低设计的复杂度。如何获得更好的模块化设计不是这里要讨论的重点，本书只关注模块的

初始化与终止化这两个关键点。

在大多的嵌入式系统中，模块的运行是从调用它的初始化函数开始的。与模块大多有初始化函数相比，忽视为模块设计终止化函数这种现象却很普遍。与运行在桌面操作系统上的软件不同的是，通常整个嵌入式设备就只有一个应用软件在运行，因此对软件启停不少会采用开关设备电源或按下重启按钮这种“粗暴的”方式来完成，久而久之大家将为模块设计终止化函数当做了多余。

首先，从完整性的角度来看，一个模块如果提供初始化函数，那么也应当设计终止化函数。其次，为每一个模块设计终止化函数，意味着提供了一种“优雅地”关闭系统的手段，进一步的内涵是，通过这种方式将为我们创造检测系统资源泄漏的时机。

让我们站在堆管理模块的角度来检查优雅终止模块所带来的好处。这里假设堆管理模块具备记录每一次内存分配所发生的位置信息这一功能。以它为例，是因为内存泄漏是嵌入式软件开发中比较让人头痛的问题。

如果一个系统中所有模块的终止行为都不经过各自的终止化函数的话，堆管理模块就无法通过它所记录的分配位置信息来了解是否存在内存泄漏问题。如果只为堆模块提供终止化函数也同样无法发现内存泄漏问题，因为其他模块可能在初始化时分配内存，且这些内存在整个软件生命周期中都需要使用。在系统终止时，如果这些内存不被各模块自行释放的话，堆管理模块无法在它的终止化函数中判断哪些内存发生了泄漏。

如果为每一个模块都设计终止化函数，就能做到更容易检测内存泄漏。如果那些在初始化函数中分配内存的模块在终止化时进行内存释放操作，且假设所有使用了动态内存的模块的终止化函数是在堆管理模块的终止化函数之前被调用的，那么当堆管理模块的终止化函数被调用时，就可以根据所记录的信息找到没有释放的内存（泄漏点）。

依此类推，其他的资源也可以采用这一方法发现泄漏。比如，在定时器管理模块的终止化函数中可以检查是否所有的定时器都已回收了。

在模块的终止化函数中检查所管理资源是否存在泄漏需要解决一个问题，即各个模块的依赖关系，以保证各资源管理模块的终止化函数是在使用它（所管理资源）的模块之后被调用的。第 14 章所引入的模块分层与分级的概念将有助于解决模块间的依赖关系。

本书操作系统篇中的多个章节采用了这一设计原则以防范资源泄漏，读者在阅读这些章节时将加深对这一设计原则的理解。

13.6.4 重视收集统计信息

统计信息在好多方面将发挥作用。首先，它可被用于软件调优。软件在开发过程中不可能完全了解现场运行环境，这导致系统可能无法运行在最佳状态（性能、可使用性等）。要让系

统处于最佳运行状态，必须获得软件在现场的运行环境，这需要通过一定的数据，设计必要的统计信息将有助于获得所需的现场数据。

其次，统计信息还有助于查错。打个比方，如果一个软件系统是从设备外部接收消息并对之进行处理和响应的，如果设计有进入系统消息数量的统计信息，那么当某些情形下出现负荷异常高时，通过它就可以判断负荷异常是由外部引起的还是由内部造成的。大多难以定位错误根源的软件缺陷，正是因为“蛛丝马迹”太少了，而统计信息有助于捕获它们。

统计信息通常采用为每一个统计项定义一个整型变量的形式，图 13.7 是一个定时器模块所设计的相关统计信息。从第 29 和 48 行可以看出，statistic_t 类型就是整型的 typedef。第 70、71 和 78～80 行则定义了相应的统计变量。当相应的情形出现时，则对对应的统计变量进行加一操作。统计信息的显示就是将统计变量的值通过一定形式输出。由此看来，增加统计信息的成本不论从内存空间还是处理器时间上其开销都是很经济的。

```
00029: typedef unsigned int u32_t;
00047: // for statistic
00048: typedef u32_t statistic_t;
00049:

00068: typedef struct {
00069:     dll_t dll_;
00070:     statistic_t hit_;
00071:     statistic_t redo_;
00072:     csize_t reentrance_;
00073:     csize_t level_;
00074: } bucket_t;
00075:
00076: typedef struct {
00077:     statistic_t notimer_;
00078:     statistic_t traversed_;
00079:     statistic_t abnormal_;
00080: } timer_statistic_t;
00081:
00082: static timer_statistic_t g_statistic;
```

图 13.7

运用这一设计原则，需要我们在设计过程中时刻思考哪些信息对软件查错和调优有帮助。统计项的设计并不要求一步到位，可以在软件生命周期的任何阶段根据需要而增加。

13.6.5　借助命名传达设计意图

程序代码是设计的物质外壳，再好的思想必须最终通过代码去表达，而这就离不开对函数、变量和参数进行恰当的命名，以便准确地传达设计意图。让我们通过一个例子来说明命名对设计的重要性。

假设需要设计一个双向链表的操作函数，其功能是删除链表头结点并将之当做函数的返回值返回，那如何给这一函数取名呢？

dll_get_head()[⑤]这个命名可能会在读者的脑海中浮现，但这个命名并不好，因为“get”所表达的是“取”而没有“删除”的意思。当他人读到对dll_get_head()函数的调用时，将理解为“只是引用头结点但并不将它从链表中删除”，这显然没有精确地传达设计本意。一个没有传达设计本意的命名是注定要让人困惑的。

那dll_extract_head()呢？同样不好，因为“extract”也不能表达从链表中删除结点的意思。用过WinZip英文版的读者或许注意到了，其中解压文件就用了“extract”这个词。

比较合适的命名是dll_pop_head()。“pop”这一动词源于退栈操作，当从栈中“弹出”一个元素时，意味着这一元素将从栈中被删除并返回。将“pop”引入双向链表的函数名中能准确地传达设计意图。

除了函数名的命名很重要外，函数参数和变量的命名同样重要，因为它们能起到“点睛”的作用。对于图13.8中的两个函数，从参数名就能完全明白如何用，任何解释用法的注释都显得多余。

```
void dll_insert_before (dll_t *_p_dll, dll_node_t *_p_ref,
   dll_node_t *_p_inserted);
void dll_insert_after (dll_t *_p_dll, dll_node_t *_p_ref,
   dll_node_t *_p_inserted);
```

图13.8

软件设计的最终产物不能是一堆难读的代码；相反，代码应当努力做到让人读起来“行云流水”。好的设计在看完它的接口函数和数据结构后就知道如何使用它，因为它们的命名向人传达了模块的行为。从这一点说来，花时间斟酌命名是值得的，因为它节省了他人用于理解的时间。

在不少资料中强调注释对于编码的重要性，甚至提出程序应有三分之一的篇幅是注释。对于“三分之一”这一提法，作者并不赞同。原因是：

- 我们在读程序时的第一反应是读代码而不是注释。如果代码能清楚地表达意思，那就没有写注释的必要，即使写了那也一定是多余。如果注释占了整个项目源程序的三分之一，作者怀疑其中很多都是废话，只是为了做到“占三分之一”而已。
- 注释与代码很容易在维护的过程中失步，因此会出现注释所发出的声音与源代码实现完全不同这种尴尬。一旦注释被发现不精确它就会被人遗忘，也就起不到注释应有的效果。

与“三分之一”提法不同的是，作者认为注释应当尽可能少，并将写注释所省下来的时间用于推敲命名。请注意，千万不要误认为“少注释是好程序的充分条件”，而应当理解为“少

⑤ 函数名中的dll是Double-Linked List，即双向链表的简写。

注释是好程序的必要条件”。

诚然，这并不是在否定注释。大部分情形下，通过命名就能清楚地表达程序实现的局部思想，而注释应当放眼于全局去写以起到提纲挈领的作用，或者某些行为打破了常规(比如，switch 语句块内的 case 与 break 不成对)也可以考虑通过注释加以解释。如果命名实在无法做到“传神”或打破了常识的话，也可以考虑采用少量的注释进行弥补。

13.6.6　消除“审美告警”

人天生就是审美家，软件工程师在进行软件设计时这种天性会自然地发挥作用。将设计感觉当做是一个设计原则，多少让人觉得有点不靠谱，但这一原则也正体现了软件设计中的艺术成分。

就作者的经验来看，如果做设计时觉得别扭，那工作效率一定不高；反之，则工作效率奇高。软件设计真正花时间的是思考，而不是编码。思考的目的是从纷繁的现象中试图找到问题的本质，或者从众多的因素中找出关键。设计时之所以出现“审美告警”，一定是有什么没有考虑清楚。这种情形下停下来做进一步的思考，将有助于理清思路，以最终获得更好的设计。

相信每个软件工程师或多或少都能感觉到“审美告警”，而信号的强弱与软件工程师的设计水平可能是正相关的。软件工程师如果重视这种信号，则这种信号的灵敏度也会慢慢提高。因为重视它意味着将进行更多的思考，而思考多了就更容易形成自己的设计原则和思想；相反，如果长期忽视它的存在，则最终可能会造成这种信号的消失。忽视“审美告警”的存在，或许意味着我们并不关心所设计的主题，其质量也别指望好到哪儿去，更有甚者会酝酿出将来的一个“毒瘤”。

13.6.7　通过机制解决问题

设计并不只是存在于全新项目的开始阶段，而可以存在于软件生命周期的任何时刻。

软件是注定要出错的，这是因为人的大脑存在局限性。当一个问题出现了以后，在寻求设计解决方案时大致存在两种方法。第一种方法就是“头痛治头，脚痛治脚”，即只针对单个问题去寻求解决方案；第二种方法则是采用设计通用机制。这种方法通常除了解决已经发生的问题外，还能预防其他类似的但还没有发生的问题。下面通过具体的例子来说明什么是通过机制解决问题。

现在假设存在两个设备，分别是 A 和 B，它们之间通过以太网进行通信。假设正常情形下存在图 13.9 所示的消息流片段，即设备 A 向设备 B 发送一个请求消息（REQ），在设备 B 收到来自 A 的请求后，经过一定的消息处理发送回应消息（RSP）以作响应，设备 A 则以确认消息（ACK）对之加以确认。这三个消息的来回就完成了一次完整的通信。

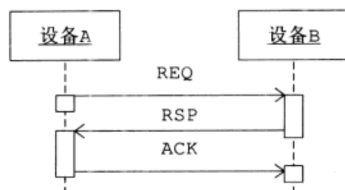


图 13.9

后面的内容假设我们站在设备 B 的角度。如果设备 B 是采用状态机的实现来处理消息的话，则将得到图 13.10 所示的状态机。其中的“REQ/发送 RSP”表示在“等待 REQ”状态如果收到 REQ 消息，则发送 RSP 消息，并迁移到“等待 ACK”状态。

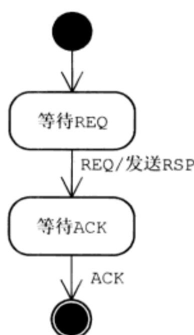


图 13.10

当设备 B 发送 RSP 消息到设备 A 后，很有可能这一消息因为丢失而导致设备 A 收不到，图 13.11 示例说明了这一情形。很容易想到，设备 A 如果没有收到来自设备 B 的 RSP 消息的话，它将采用重发 REQ 消息的方式进行尝试。

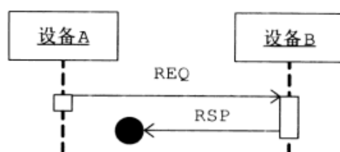


图 13.11

但是，设备 B 的状态机在发送完了 RSP 消息后，立即迁移到了“等待 ACK”状态，在这一状态下由于状态机并不理会 REQ 消息，所以即使设备 A 重发 REQ 消息也不会得到设备 B 的 RSP 回应消息。解决这一问题最为直接的办法是更改状态机的实现，如图 13.12 所示，即在“等待 ACK”状态下增加对 REQ 消息的处理。

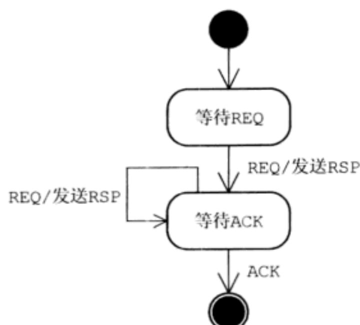


图 13.12

这一方法乍一看能解决问题，但是它增加了设备 B 软件实现的复杂度。试想一想，如果存在大量的状态需要迁移到“等待 ACK”状态，则需要在“等待 ACK”状态中增加大量的应在前一状态处理的消息，如此一来“等待 ACK”状态的实现会更复杂。这一解决方法有“脚痛治脚”的嫌疑。

通过机制解决问题的方案如图 13.13 所示，即在应用层之下建立了一个新的抽象层——消息缓存与重发层。

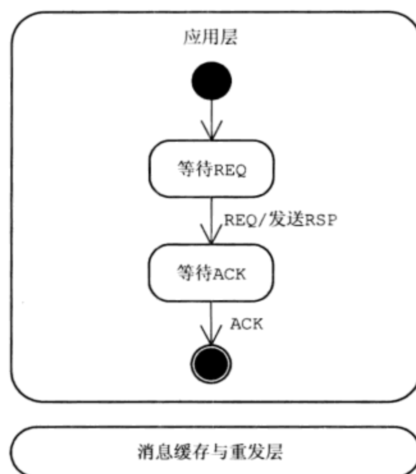


图 13.13

思路是，所有驱动状态机的消息都应当经由消息缓存与重发层发送给应用层，当应用层处理完了一个消息并且需要发送回应消息时也必须经过消息缓存与重发层发送出去。消息缓存与重发层在收到来自应用层发出的回应消息时，会建立收发消息间的对应关系。

当消息缓存与重发层收到一个消息时，先查看所记录的映射信息中是否存在一个回应消息与之对应，如果有则将它直接发送出去且不用将收到的消息转给应用层；如果不存在则将收到

的消息转给应用层去处理。

这一设计方案所带来的好处是明显的，应用层不需要考虑由于消息丢失而造成的消息重传问题，进而简化了应用层状态机的实现。这里有一个假设值得一提，消息缓存与重发层能从收到的消息确定消息是否是重传的。大多协议都有序列号或交易号等信息，它们可被用于标识消息。

13.6.8　防止他人犯错

在设计过程中不能一味地站在设计者的角度，还得站在模块使用者的角度去思考。设计者容易假设使用者是完全沿着自己的设计思路去使用模块的，这种假设在现实中表现得很脆弱。问题的发生不少情形是因为使用者打破了设计者原有的假设，即使被设计模块的使用者是本人，也仍需要站在使用者的角度去指导设计。下面看一个相关的例子。

假设被设计的定时器模块是通过内存分配的方式创建定时器实例的，简化了的示例程序可以从图 13.14 中找到。

```
example.c
00300: typedef struct {
00301:     expiry_callback_t callback_;
00302:     char name_ [TIMER_NAME_MAX + 1];
00303:     ......
00304: } timer_t, *timer_handle_t;
00305:
00306: timer_t timer_alloc (timer_handle_t *_p_handle, msecond_t _duration,
00307:     expiry_callback_t _callback, const char *_name, void *_p_arg)
00308: {
00309:     timer_t *p_timer = malloc (sizeof (timer_t));
00310:     ......
00311:     return p_timer;
00312: }
00313:
00314: error_t timer_free (timer_t *_p_timer)
00315: {
00316:     ......
00317:     free (_p_timer);
00318:     return 0;
00319: }
00320:
00321: void timer_fire ()
00322: {
00323:     timer_t *p_timer;
00324:     ......
00325:     p_timer->callback_(p_timer, p_timer->arg_);
00326:     ......
00327: }
```

图 13.14

图中，timer_alloc()函数在第 309 行调用 malloc()函数分配内存以创建一个定时器实例，并将实例指针返回给用户。当定时器到期时 timer_alloc()函数的参数_callback 所指定的回调函数将会被调用，这一调用动作发生在 timer_fire()函数的第 325 行。timer_free()函数被用于释放定时器，它需要在第 317 行调用 free()函数释放第 309 行分配获得的内存。

在这一实现中，如果定时器模块的使用者在定时器的回调函数中删除定时器自己，且 timer_fire()函数的第 325 行之后又使用 p_timer 指针就会导致程序崩溃。预防这一问题最简单的方法是增加一个限制条件，即规定在定时器的回调函数中不能删除自己，但这不是最好的方法。更好的方法需要通过设计去解决，图 13.15 示例说明了一种设计实现。

```
00300: typedef struct {
00301:     expiry_cb_t callback_;
00302:     char name_ [TIMER_NAME_MAX + 1];
00303:     bool deletion_pending_;
00304:     bool is_in_callback_;
00305:     ......
00306: } timer_t, *timer_handle_t;
00307:
00316: error_t timer_free (timer_t *_p_timer)
00317: {
00318:     ......
00319:     if (_p_timer->is_in_callback_) {
00320:         _p_timer->deletion_pending_ = true;
00321:     }
00322:     else {
00323:         free (_p_timer);
00324:     }
00325:     return 0;
00326: }
00327:
00328: void timer_fire ()
00329: {
00330:     timer_t *p_timer;
00331:     ......
00332:     p_timer->is_in_callback_ = true;
00333:     p_timer->callback_(p_timer, p_timer->arg_);
00334:     p_timer->is_in_callback_ = false;
00335:     ......
00336:     if (p_timer->deletion_pending_) {
00337:         free (p_timer);
00338:     }
00339: }
```

图 13.15

在新实现中，timer_t 数据结构中增加了 deletion_pending_和 is_in_callback_两个变量。is_in_callback_变量的作用就是用于指示定时器的回调函数是否正被调用，而 deletion_pending_的作用就是指示是否在回调函数中调用了 timer_free()函数。当在回调函数中调用了 timer_free()

函数时，我们并不立即进行内存释放操作，而是通过 deletion_pending 变量进行标识，表示这一定时器需要删除。最后在完成回调函数的执行后，在第 336 行检查 deletion_pending_变量看是否需要进行内存释放操作。

在新的实现中，我们完全不需要对定时器模块施加“在定时器的回调函数中不能删除自身”这样的限制，这也意味着能防范他人因违背限制而发生的错误。

13.6.9 考虑可查错性

查错性设计是指当错误发生时我们如何来做到更好地定位问题范围。下面通过例子来看一看这一设计原则是如何被运用的。

图 13.16 示例说明了运行在 Linux 操作系统上某项目的一个类图。图中的“SNMP Agent”进程正是作者所在的项目团队开发的，而 Box 库是由位于其他国家的团队开发的。FDR(Function Definition Rule)是指一套函数定义规则，它作为“SNMP Agent”进程与库之间的接口。

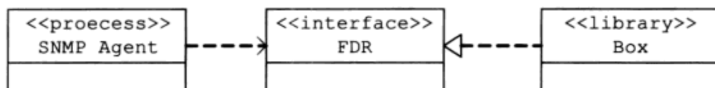


图 13.16

“SNMP Agent”进程实现的功能是使得设备之外的网络管理程序可以通过“SNMP Agent”进程提供的 SNMP 接口来管理电信设备。“SNMP Agent”进程收到设备外部发送过来的 SNMP 消息之后，将消息最后转换为函数调用的形式来存取信息，而这些函数的实现是位于 Box 库中的。由于 Box 库是被“SNMP Agent”进程调用的，因此一旦 Box 库中存在严重的缺陷，最终将直接导致“SNMP Agent”进程崩溃。

对于不少崩溃的情形，通过运用 gdb 工具查看崩溃时的调用栈能很快地找到问题的根源，这类问题并不算大。但是，在使用 gdb 工具查看调用栈时，如果发现任务的堆栈被破坏了（stack corrupted）那就麻烦了，这种错误也是行业内最难解决的问题。

一旦“SNMP Agent”进程出现了崩溃，第一反应是让“SNMP Agent”的开发团队去排错。而此时如果问题是出在 Box 库上时，无论“SNMP Agent”团队如何努力都查不出问题的根源，最终可能导致问题被搁置。为了防止这类问题因为定位难而搁置，我们就需要从设计上进行防范。

作者的思路是，设计一种方法记录“SNMP Agent”进程正在运行哪一部分代码。当然，信息的粒度只要足于区分是运行 Box 库中的代码还是“SNMP Agent”的代码就够了。

很直接的做法是，每次调用 Box 库中的函数时，都在调用之前打印一行日志，在调用返回

后又打印一行日志，但这种方式并不经济。可以通过使用 Linux 中的共享内存来提高程序的效率。Linux 中的共享内存有一个特点，当应用程序分配了一块共享内存，只要不主动将其删除，那么即使程序出现了崩溃，其中的内容也一直保存在那不会被更改，且其内容可以被重启后的应用程序再次访问。当然，操作系统被重启这种情形除外。

通过设计可以将调用 Box 函数之前和之后这一信息通过使用一个整型变量的方式进行表达，并将这个整型变量放入"SNMP Agent"进程所创建的共享内存中。比如设置整型值 3 表示将要调用 Box 库中的某一个函数，而在调用完了后将值设置成 4。当"SNMP Agent"进程出现崩溃后，在程序下一次启动的初始化阶段，通过共享内存中的值就可以知道上一次程序出错时，是否出现在调用 Box 库函数期间，如果是则记录一条错误日志。具体地，如果发现共享内存中整型变量的值为 3，则说明崩溃发生在 Box 库内。有了这种方法后，我们就可以通过日志很好地界定出现问题的模块了。

这一设计原则有时会使得程序实现更加复杂，评估所引入的复杂度是否值得不能仅从技术层面进行考虑，还需从系统和团队的角度去思考。如果模块分属两个团队，且问题难于界定而造成团队之间相互推诿的话，通过设计加以避免是很不错的选择。

13.7　小结

设计是软件质量之本。产品的主导设计一旦确定了，其质量水准也随之确定。不良设计无论通过怎样的测试，都无法实现团队级的高质量。

项目因为不良设计而进入混乱状态并不可怕，只要我们拥有一颗拥抱变化的心。通过运用"乱中求治"的思想，不仅能有效控制改善设计所带来的风险，还能锻炼团队并最终走出设计困境。

好设计之所以好，是因为它符合一定的设计原则。我们需要通过掌握设计原则去提高自己的设计能力，设计能力的提高应以形成自己的设计思想为终极目标。

本篇的后面几章，我们将一同探讨几个最佳设计实现和一些设计思想。

第 14 章 模块管理，保障系统有序运行

采用模块化的方法设计软件，早已成为行业的共识。模块化除了反映在设计方法上外，在实现上也得下足工夫，既突显模块的概念又实现整个系统的有序运行。

在 13.6.3 节中指出，应为每一个模块设计终止化函数以实现系统的优雅关闭，但这需要管理好模块间的依赖关系，并通过通用机制来控制各模块的终止化顺序。本章将介绍一种模块管理的通用机制，该机制也被运用到本书操作系统篇中任何需要的地方。

14.1 管理参照系

当一个系统比较复杂，所包含的模块数量比较多时，不可避免地会产生模块间复杂的依赖关系，且有可能出现“牵一发而动全身”这种情形。无论采用怎样的方法管理模块，都应当先将系统中所有模块间的依赖关系通过某种方式表达出来，而 UML 中的类图是一个不错的选择。

先看一看图 14.1 所示的某系统各模块间的依赖关系。从依赖关系可以看出，当系统进行初始化时堆管理模块应当最先被初始化，接着是紧跟其后的定时器管理模块和内存池管理模块。图中所有模块的初始化顺序是从上到下的，且左右顺序并不重要。

为了更好地表达模块间的依赖关系，我们需要引入分层的概念。一个系统中的模块可以分成三大层，分别是平台层、框架层和应用层。对于图 14.1 如果采用分层的方式进行分割，就得到了图 14.2。图中用粗横线对各层进行了分割。

对系统进行初始化的顺序显然应该是平台层最先，应用层最后。而终止化应采用完全相反的顺序，即先对应用层进行终止化，最后对平台层进行终止化。这遵循了顺序分配、逆序释放的原则。

当在系统中新增加一个模块时，首先需要确定将其划入哪一个层次。一个模块是否属于应用层相对容易判断，只要看这个模块实现的功能是否是产品所私有的。如果是，那说明该模块应属于应用层，否则就属于平台层或框架层。阅读第 17 章对于判断是平台层还是框架层会有所帮助。有时一些模块的归属并不那么明显，在这种情形下，作者建议一开始不要太计较，而

是先给它定一个层次，等到项目开展下去对之认识加深时再调整也不迟。

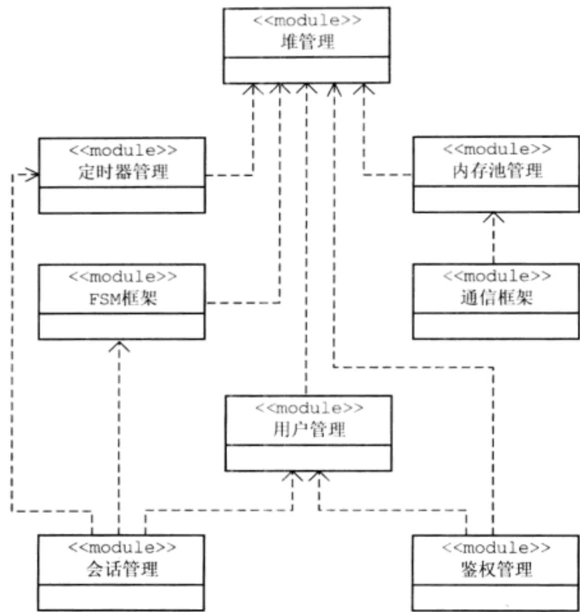


图 14.1

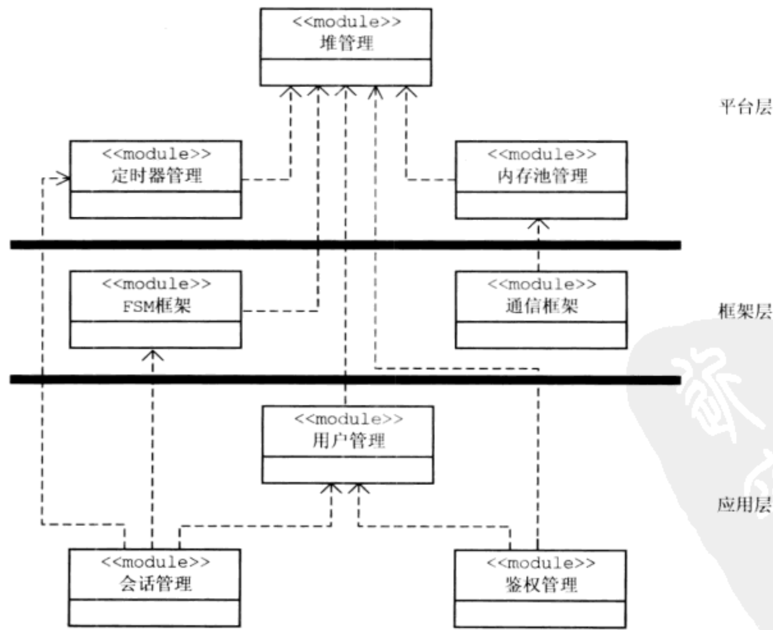


图 14.2

模块管理如果只有层的概念则粒度太大，因为在一个层中可能存在多个模块且模块之间也存

在依赖关系。比如，图 14.2 所示的平台层就存在定时器管理模块依赖于堆管理模块。由此看来在同一层中还得进行划分，为此需要引入分级的概念。图 14.3 中用细横线对同一层中的不同级进行了分割。每一层中自上而下我们将其定义为第 0 级、第 1 级，依此类推。同级可以有多个模块。

有了分层和分级的概念后，就为模块的初始化和终止化顺序提供了参照系。当新加入一个模块时，需要通过先分层后分级的方式以确定它应位于系统初始化过程中的哪一个点，这又间接地确立了终止化顺序点。在这个参照系中，模块的初始化顺序只与依赖关系图中的上下位置有关，而与左右顺序无关。如果左右之间的模块在某种情形下存在依赖关系，那就要把它们划分到不同的级而转化为上下关系。

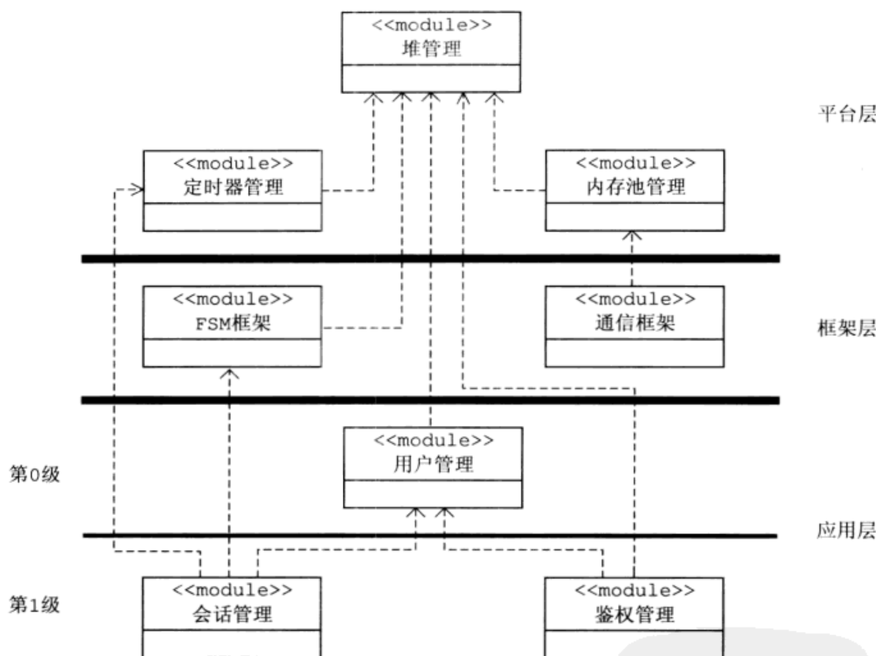


图 14.3

14.2 设计思路

模块管理最主要是要做到能集中控制所有模块的初始化和终止化顺序，而要实现这个设计目的，方式之一是让每一个模块通过注册回调函数的形式实现控制反转。图 14.4 的顺序图示例说明了模块管理模块的外部行为。

如图中所示，每一个模块都需要调用 module_register()函数向模块管理模块进行注册，注册时需要指明模块所属层级和模块的回调函数。当系统启动时，system_up()函数被调用，这时会直接触发模块管理模块根据层与级的顺序调用各注册模块的回调函数实现模块的初始化。反

之，当系统终止时，system_down()函数被调用，它将触发各模块进行终止化操作，终止化操作的顺序与初始化过程是完全相反的。

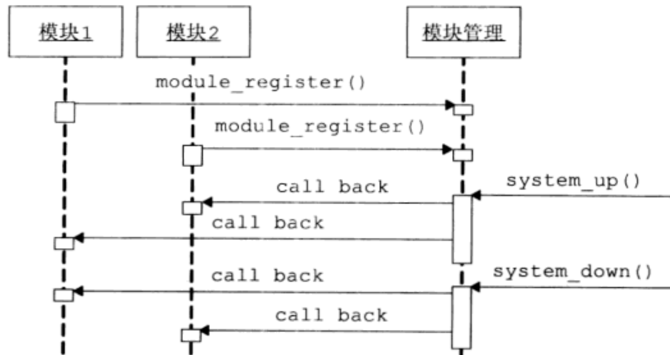


图 14.4

从追求简单性的原则考虑，每一个注册模块只需注册一个回调函数。回调函数被设计成包含一个参数以指示每一次被调用时的目的。这里我们定义了“初始化(Initializing)”、“已启动(Up)”、“已停止(Down)”和“终止化(Destroying)”4 个系统状态，每个模块的回调函数以它们作为参数值。图 14.5 示例说明了系统状态的迁移。

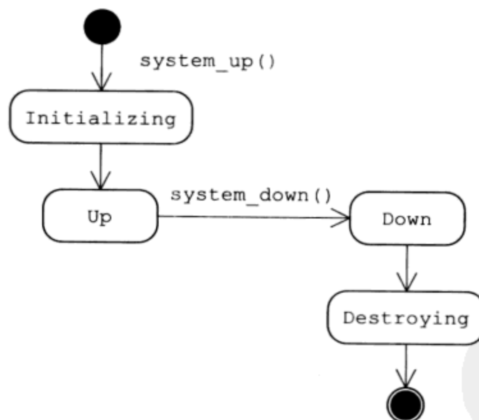


图 14.5

注意：这里定义 4 个状态而非 2 个状态是考虑到——“初始化”和“已启动”是针对系统启动时的，“已停止”和“终止化”则是针对系统终止时的，也就是说，为每个模块分别提供两次初始化和终止化的机会，通过这样的定义有助于更好地管理模块间的依赖关系。比如，各模块可以在“初始化”状态做只依赖于低层模块的初始化工作，而在“已启动”状态可以做依赖其他高层模块的初始化工作。

14.3　程序实现

至此，我们已经塑造了模型，接下来看一看具体实现。

14.3.1　引入模块标识

要实现模块管理需要为模块定义数据结构。首先，需要引入模块标识值以方便识别和管理每个模块。标识值从 0 开始，采用这种方式有助于使用数组这种简单的数据结构实现模块管理，使得模块标识值可用做数组下标。图 14.6 示例说明了 embedded 项目中所定义的模块标识。

```
00031: typedef enum {
00032:     MODULE_MODULE,       // for module management
00033:     MODULE_INTERRUPT,    // for interrupt management
00034:     MODULE_DEVICE,       // for device management
00035:     MODULE_CLOCK,        // for clock management
00036:     MODULE_CONSOLE,      // for device console
00037:     MODULE_CTRLC,        // for handling Ctrl+C on Linux/Cygwin
00038:     MODULE_FLASH,        // for device of flash
00039:     MODULE_TIMER,        // for timer management
00040:     MODULE_TASK,         // for task
00041:     MODULE_SYNC,         // for task sync object management
00042:     MODULE_SEMAPHORE,    // for semaphore management
00043:     MODULE_MUTEX,        // for mutex management
00044:     MODULE_QUEUE,        // for queue management
00045:     MODULE_HEAP,         // for heap management
00046:     MODULE_MPOOL,        // for memory pool management
00047:
00048:     MODULE_TESTAPP,      // for Test Application
00049:
00050:     // !!! NOTE: please always put the MODULE_COUNT and MODULE_LAST
00051:     // at the end of this enum
00052:     MODULE_COUNT,
00053:     MODULE_LAST = (MODULE_COUNT - 1)
00054: } module_t;
```

图 14.6

其中，MODULE_COUNT 和 MODULE_LAST 并不对应相应的模块。MODULE_COUNT 用于表示整个系统一共有多少个模块，而 MODULE_LAST 表示系统中最后一个模块的标识值。当需要增加新模块标识值时，在枚举体内直接添加即可，但必须添加在 MODULE_COUNT 之前，这样新模块标识值的添加并不会影响 MODULE_COUNT 和 MODULE_LAST 所表达的意思。

模块标识除了被运用于模块管理外，在第 15 章还展示了如何将其运用到错误码中，以标识一个错误发生在哪一个模块。

14.3.2　实现层与级的表达

尽管在模块管理参照系中，层与级是完全不同的概念，但从实现的角度来看，这两个概念

的目的却都是为了表达模块初始化和终止化时的先后顺序。因此，在程序实现中完全可以对它们采用同样的方法加以表达，但通过名称加以区分。图 14.7 定义了 init_level_t 数据类型以表达层与级的概念。

```
00056: typedef enum {
00057:     LEVEL_FIRST,
00058:     CPU_LEVEL = LEVEL_FIRST,
00059:     PERIPHERALS_LEVEL,
00060:     DRIVER_LEVEL,
00061:     OS_LEVEL,
00062:
00063:     // for platform layer
00064:     PLATFORM_LEVEL0,
00065:     PLATFORM_LEVEL1,
00066:     PLATFORM_LEVEL2,
00067:     PLATFORM_LEVEL3,
00068:     PLATFORM_LEVEL4,
00069:     PLATFORM_LEVEL5,
00070:     PLATFORM_LEVEL6,
00071:     PLATFORM_LEVEL7,
00072:
00073:     // for framework layer
00074:     FRAMEWORK_LEVEL0,
00075:     FRAMEWORK_LEVEL1,
00076:     FRAMEWORK_LEVEL2,
00077:     FRAMEWORK_LEVEL3,
00078:     FRAMEWORK_LEVEL4,
00079:     FRAMEWORK_LEVEL5,
00080:     FRAMEWORK_LEVEL6,
00081:     FRAMEWORK_LEVEL7,
00082:
00083:     // for application layer
00084:     APPLICATION_LEVEL0,
00085:     APPLICATION_LEVEL1,
00086:     APPLICATION_LEVEL2,
00087:     APPLICATION_LEVEL3,
00088:     APPLICATION_LEVEL4,
00089:     APPLICATION_LEVEL5,
00090:     APPLICATION_LEVEL6,
00091:     APPLICATION_LEVEL7,
00092:
00093:     // LEVEL_COUNT and LEVEL_LAST must be put at the end of this enum
00094:     LEVEL_COUNT,
00095:     LEVEL_LAST = (LEVEL_COUNT - 1)
00096: } init_level_t;
```

图 14.7

从图中的枚举内容来看，其中包括平台（PLATFORM）、框架（FRAMEWORK）和应用（APPLICATION）三大层，且每一层又定义了八个级，分别从 LEVEL0 到 LEVEL7。除了这三大层八个大级外，还额外定义了处理器级（CPU_LEVEL）、外设级（PERIPHERALS_LEVEL）、驱动级（DRIVER_LEVEL）和操作系统级（OS_LEVEL）。从某种意义上来说，这四个级可以被归类为平台层，将它们独立出来完全是为了使概念更清晰。

14.3.3 系统状态和回调函数原型定义

图 14.8 中定义了用于表示系统四个状态的 system_state_t 数据类型和模块回调函数原型 moudle_callback_t。正如前面所提到的，模块回调函数是以系统状态为参数的。

```
00098: typedef enum {
00099:     STATE_INITIALIZING,
00100:     STATE_UP,
00101:     STATE_DOWN,
00102:     STATE_DESTROYING
00103: } system_state_t;
00104:
00105: typedef error_t (*module_callback_t)(system_state_t _state);
```

图 14.8

14.3.4 模块注册

图 14.9 示例说明了用于管理模块所需的数据结构和全局变量。

```
00033: typedef struct {
00034:     dll_node_t node_;
00035:     const char *p_name_;
00036:     module_callback_t callback_;
00037:     bool is_registered_;
00038: } module_init_t;
00039:
00040: static dll_t g_levels [LEVEL_COUNT];
00041: static module_init_t g_modules [MODULE_COUNT];
```

图 14.9

每一个模块需要使用一个 module_init_t 结构实例记录它的注册信息。module_init_t 结构各成员变量的作用是：

- node_：当用户进行模块注册时，会通过这个链表结点将相同初始化级别的模块串接在一起。
- p_name_：用于记录模块名。这里存在一个假设，即所传入的模块名的内存并不是采用 malloc()函数分配的，而是采用字符串字面值的形式传入的。这一假设使得可以省去内存拷贝而简化实现。
- callback_：用于保存模块的回调函数，系统状态的变迁都将导致各模块的回调函数被依次调用。
- is_registered_：这个变量用于防止模块的重复注册。当一个模块被注册后，这个变量的值将变成 true。

第 40 行的数组 g_levels 为每一个初始化级别定义了一个链表。图 14.10 示例说明了链表数

据结构的定义。当模块被注册到某一级别时，模块的注册信（即 module_init_t 数据结构实例）将挂接到相应的链表上（同一级中的多个模块我们以链表的方式进行管理）。第 41 行的数组 g_modules 则为每一个模块定义了一个存放其注册信息的数据实体。

```
embedded/code/platform/common/src/dll.h
00031: /* DLL stands for Double-Linked List */
00032:
00033: typedef struct dll_node {
00034:     struct dll_node *prev_;
00035:     struct dll_node *next_;
00036: } dll_node_t, *dll_node_handle_t;
00037:
00038: typedef struct {
00039:     dll_node_t *head_;
00040:     dll_node_t *tail_;
00041:     usize_t count_;
00042: } dll_t, *dll_handle_t;
```

图 14.10

module_register()函数用来实现模块注册，其实现如图 14.11 所示。该函数各参数的含义是：参数_name 指明被注册模块的名称是什么；参数_module 指示所需注册的模块标识值；参数_level 标明这一模块将属于哪一个初始化级别；参数_callback 用于指示模块的回调函数。

```
embedded/code/platform/common/src/module.c
00044: error_t module_register (const char _name [], module_t _module,
00045:     init_level_t _level, module_callback_t _callback)
00046: {
00047:     module_init_t *p_module;
00048:
00049:     if (_module > MODULE_LAST) {
00050:         return ERROR_T (ERROR_MODULE_REG_INVMODULE);
00051:     }
00052:     if (_level > LEVEL_LAST) {
00053:         return ERROR_T (ERROR_MODULE_REG_INVLEVEL);
00054:     }
00055:     if (null == _callback) {
00056:         return ERROR_T (ERROR_MODULE_REG_INVCB);
00057:     }
00058:
00059:     p_module = &g_modules [_module];
00060:     if (p_module->is_registered_) {
00061:         return ERROR_T (ERROR_MODULE_REGISTERED);
00062:     }
00063:     p_module->p_name_ = _name;
00064:     p_module->callback_ = _callback;
00065:     p_module->is_registered_ = true;
00066:     dll_push_tail (&g_levels [_level], &p_module->node_);
00067:
00068:     return 0;
00069: }
```

图 14.11

第 49～57 行检查输入的参数是否有效。第 59 行获取管理模块所需的数据结构实例。第 60 行则检查模块是否已注册过。第 63 和 64 行分别记录模块的名称和回调函数。第 65 行将

is_registered_变量的值置成 true，表示模块已被注册过。第 66 行则将模块信息挂接到被注册级别的链表中。

注意：这个注册函数并没有采用上锁的方式以防止出现竞争问题，原因是模块注册行为通常发生在系统的最开始阶段，而此时并不存在多任务问题。别忘了，任务的创建是在注册模块的回调函数中完成的。

从模块注册函数的实现来看，它只是告知了被注册模块处于什么级别，而真正的初始化或终止化操作动作并没有发生。这两个行为是通过调用 system_up()和 system_down()函数触发的。

14.3.5 系统启动

system_up()函数的调用标志着系统开始启动，其实现如图 14.12 所示。

```
embedded/code/platform/common/src/module.c
00042: static system_state_t g_state;
00043:
00071: static bool init_for_each (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00072: {
00073:     module_init_t *p_module = (module_init_t *)_p_node;
00074:     error_t result = p_module->callback_ (STATE_INITIALIZING);
00075:
00076:     UNUSED (_p_dll);
00077:     UNUSED (_p_arg);
00078:
00079:     if (0 != result) {
00080:         console_print ("Error: can't initialize module %s (%s))",
00081:             p_module->p_name_, errstr (result));
00082:         return false;
00083:     }
00084:     return true;
00085: }
00086:
00087: static bool up_for_each (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00088: {
00089:     module_init_t *p_module = (module_init_t *)_p_node;
00090:     error_t result = p_module->callback_ (STATE_UP);
00091:
00092:     UNUSED (_p_dll);
00093:     UNUSED (_p_arg);
00094:
00095:     if (0 != result) {
00096:         console_print ("Error: can't start up module %s (%s))",
00097:             p_module->p_name_, errstr (result));
00098:         return false;
00099:     }
00100:     return true;
00101: }
00102:
00103: error_t system_up ()
00104: {
00105:     init_level_t level;
00106:
00107:     g_state = STATE_INITIALIZING;
00108:     for (level = LEVEL_FIRST; level <= LEVEL_LAST; ++ level) {
00109:         if (0 != dll_traverse (&g_levels [level], init_for_each,
```

```
00110:            (void *)&level)) {
00111:            return ERROR_T (ERROR_MODULE_INIT_FAILURE);
00112:        }
00113:    }
00114:
00115:    g_state = STATE_UP;
00116:    for (level = LEVEL_FIRST; level <= LEVEL_LAST; ++ level) {
00117:        if (0 != dll_traverse (&g_levels [level], up_for_each,
00118:            (void *)&level)) {
00119:            return ERROR_T (ERROR_MODULE_UP_FAILURE);
00120:        }
00121:    }
00122:
00123:    return 0;
00124: }
```

图 14.12

system_up()函数将从 LEVEL_FIRST 级开始，依次按序调用 g_levels 数组中各链表内所记录模块的回调函数，对于链表中各结点的遍历是通过调用 dll_traverse()函数实现的。

第 107～113 行是在 STATE_INITIALIZING 状态下调用各注册模块的回调函数。遍历链表的结点回调函数是 init_for_each()，其实现位于第 71～85 行，它在第 74 行调用模块的回调函数，如果回调函数返回错误，则在第 80 行输出错误日志并让函数返回 false。init_for_each()函数一旦返回 false，就会使得第 109 行的 dll_traverse()函数终止后续结点的遍历，并返回出错模块的结点指针。在这种情形下，system_up()函数在第 111 行返回错误，而这将最终导致整个系统启动失败。

第 115～121 行与前面的实现很相似，只是此时的系统状态为 STATE_UP，遍历链表的结点回调函数也变成了 up_for_each()。

值得一提的是，在 system_up()函数内通过使用 LEVEL_FIRST 和 LEVEL_LAST 两个枚举值，使得 init_level_t 数据类型内的级别即使被更改也不必跟着更改，这提高了程序的可维护性。

为了方便理解和阅读，下面结出链表遍历函数 dll_traverse()的实现，如图 14.13 所示。

```
00158: dll_node_t *dll_traverse (dll_t *_p_dll, traverse_callback_t _cb, void *_p_arg)
00159: {
00160:     register dll_node_t *p_node = _p_dll->head_;
00161:
00162:     if (null == _cb) {
00163:         return 0;
00164:     }
00165:
00166:     while ((0 != p_node) && ((*_cb) (_p_dll, p_node, _p_arg))) {
00167:         p_node = p_node->next_;
00168:     }
00169:
00170:     return p_node;
00171: }
```

图 14.13

14.3.6 系统关闭

系统关闭函数 system_down()的实现与 system_up()很相似，它的代码如图 14.14 所示。

```
embedded/code/platform/common/src/module.c
00126: static bool down_for_each (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00127: {
00128:     module_init_t *p_module = (module_init_t *)_p_node;
00129:     error_t result = p_module->callback_ (STATE_DOWN);
00130:
00131:     UNUSED (_p_dll);
00132:     UNUSED (_p_arg);
00133:
00134:     if (0 != result) {
00135:         console_print ("Error: can't shut down module \"%s\" (%s)",
00136:             p_module->p_name_, errstr (result));
00137:         // !!! don't return false
00138:     }
00139:     return true;
00140: }
00141:
00142: static bool destroy_for_each (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00143: {
00144:     module_init_t *p_module = (module_init_t *)_p_node;
00145:     error_t result = p_module->callback_ (STATE_DESTROYING);
00146:
00147:     UNUSED (_p_dll);
00148:     UNUSED (_p_arg);
00149:
00150:     if (0 != result) {
00151:         console_print ("Error: can't destroy module \"%s\" (%s)",
00152:             p_module->p_name_, errstr (result));
00153:         // !!! don't return false
00154:     }
00155:     return true;
00156: }
00157:
00158: void system_down ()
00159: {
00160:     init_level_t level;
00161:
00162:     g_state = STATE_DOWN;
00163:     for (level = LEVEL_LAST; level > LEVEL_FIRST; -- level) {
00164:         (void) dll_traverse_reversely (&g_levels [level], down_for_each, null);
00165:     }
00166:     (void) dll_traverse_reversely (&g_levels [LEVEL_FIRST], down_for_each, null);
00167:
00168:     g_state = STATE_DESTROYING;
00169:     for (level = LEVEL_LAST; level > LEVEL_FIRST; -- level) {
00170:         (void) dll_traverse_reversely (&g_levels [level], destroy_for_each, null);
00171:     }
00172:     (void) dll_traverse_reversely (&g_levels [LEVEL_FIRST], destroy_for_each, null);
00173: }
```

图 14.14

system_down()函数的实现与 system_up()函数有几个不同点。第一个不同点是，链表结点

回调函数 destroy_for_each()和 down_for_each()在发现错误时并不返回错误，而只是输出一行错误信息。这一行为背后的思路是，在终止化时即使出错也应当保证整个系统所有模块的终止行为都发生。第二个不同点是，system_down()函数调用各模块的注册回调函数的顺序与 system_up()函数是逆序的，它是从 LEVEL_LAST 开始到 LEVEL_FIRST。对于链表的遍历，使用的是 dll_traverse_reversely()函数而不是 dll_traverse()，即以逆序遍历链表。该函数的实现如图 14.15 所示。

```
00173: dll_node_t *dll_traverse_reversely (dll_t *_p_dll, traverse_callback_t _cb,
00174:     void *_p_arg)
00175: {
00176:     register dll_node_t *p_node = _p_dll->tail_;
00177:
00178:     if (null == _cb) {
00179:         return 0;
00180:     }
00181:
00182:     while ((0 != p_node) && ((*_cb) (_p_dll, p_node, _p_arg))) {
00183:         p_node = p_node->prev_;
00184:     }
00185:
00186:     return p_node;
00187: }
```

图 14.15

对于 system_down()函数的调用可以考虑通过人为触发的方式，比如通过命令行，或者通过以太网向嵌入式设备发送一个消息，等等。在各模块实现其终止行为时，要考虑在进行真正的终止动作之前做必要检查，以确保它所管理的资源都被回收了。如果发现仍有资源没有被回收，可以通过日志等形式提示以帮助发现潜在的资源泄漏问题。

14.4　module 示例程序

图 14.16 是一个用于测试模块管理功能的小程序，在第 66 行定义了一个 module_registration_entry()函数用于各模块的注册，模拟的两个模块的注册行为分别在第 68 和 69 行发生。

```
00026: #include <stdio.h>
00027: #include <stdarg.h>
00028: #include "module.h"
00029:
00030: error_t module_timer (system_state_t _state)
00031: {
00032:     if (STATE_INITIALIZING == _state) {
00033:         printf (" Info: timer module is initializing\n");
00034:     }
00035:     else if (STATE_UP == _state) {
00036:         printf (" Info: timer module is up\n");
00037:     }
00038:     else if (STATE_DOWN == _state) {
```

```
00039:         printf (" Info: timer module is down\n");
00040:     }
00041:     else if (STATE_DESTROYING == _state) {
00042:         printf (" Info: timer module is destroying\n");
00043:     }
00044:
00045:     return 0;
00046: }
00047:
00048: error_t module_memory (system_state_t _state)
00049: {
00050:     if (STATE_INITIALIZING == _state) {
00051:         printf (" Info: memory module is initializing\n");
00052:     }
00053:     else if (STATE_UP == _state) {
00054:         printf (" Info: memory module is up\n");
00055:     }
00056:     else if (STATE_DOWN == _state) {
00057:         printf (" Info: memory module is down\n");
00058:     }
00059:     else if (STATE_DESTROYING == _state) {
00060:         printf (" Info: memory module is destroying\n");
00061:     }
00062:
00063:     return 0;
00064: }
00065:
00066: void module_registration_entry ()
00067: {
00068:     (void) module_register ("Timer", MODULE_TIMER, OS_LEVEL, module_timer);
00069:     (void) module_register ("Memory", MODULE_HEAP, OS_LEVEL, module_memory);
00070: }
00071:
00072: int main ()
00073: {
00074:     module_registration_entry ();
00075:
00076:     printf ("\nSystem is going to be up\n");
00077:     if (0 != system_up ()) {
00078:         printf ("Error: system cannot be up\n");
00079:         return -1;
00080:     }
00081:
00082:     printf ("\nSystem is going to be down\n");
00083:     system_down ();
00084:
00085:     return 0;
00086: }
```

图 14.16

请注意 module_timer()和 module_memory()两个函数的实现形式。每个模块注册函数都通过判断所传入的系统状态以决定进行怎样的处理。测试程序的运行结果如图 14.17 所示。

```
$ make
$ ./release/module.exe
```

```
System is going to be up
  Info: timer module is initializing
  Info: memory module is initializing
  Info: timer module is up
  Info: memory module is up

System is going to be down
  Info: memory module is down
  Info: timer module is down
  Info: memory module is destroying
  Info: timer module is destroying
```

图 14.17

14.5　模块管理的一些思考

系统的启动是从模块的初始化开始的，如果某一个模块需要分配或释放资源，则这些动作都应出现在模块回调函数中，只不过两个动作是在不同的系统状态完成的。对于任务创建这类动作，也应当直接或间接地出现在模块的回调函数中。

模块管理很重要的一个目的是为了实现系统的优雅关闭，因为我们希望通过优雅关闭这一方式试图发现潜在的资源泄漏问题。要做到优雅关闭，得考虑在终止化时优雅地终止模块所创建的任务，千万不要采用直接删除任务这种粗暴的方式，20.14.3 节解释了为什么。

再提醒一次，在模块的终止化过程中检查所管理的资源是否已完全回收是非常有意义的事情，它就像一个“保镖”时刻关注着系统是否存在潜在的资源泄漏问题。

14.6　小结

通过引入分层和分级的概念，为模块的初始化和终止化提供了一个运行先后顺序的参照系，每一个模块都在这个参照系中占有一席之地。

模块管理的目的是为了做到系统中各模块有序地启停，并在系统中强化模块的概念。在设计一个模块时，需要考虑在初始化过程中获取资源（包括任务创建），以及在终止化过程中释放资源。



练习与思考

在图 14.9 所定义的 module_init_t 结构中，假设 p_name_成员变量的内存不是通过调用 malloc()从堆中分配的。如果不小心打破了这一假设，会有什么副作用？如何运用防止他人犯错这一设计原则应对假设被打破？

第 15 章 错误管理，不可或缺的用户需求

软件开发如果只需考虑一切正常的情形，那开发活动将会省时省心又省力，但这显然是不现实的。出色的软件产品，一定具有良好的出错处理能力。

工程师在日常开发活动中，不论是使用现成的库函数还是实现自己的函数，都得关心函数的错误处理。但在实际的开发活动中，错误处理却经常被忽视。

错误处理被认为是“可有可无”的功能而未被当做是显式用户需求，是造成它被忽视的重要原因。表面上，错误管理是软件的内部行为，似乎与用户无关。但是，用户一定希望获得的软件产品是一个稳定高质的软件产品，那么错误管理就是必不可少的了。

要在软件产品中落实错误管理，可以从三方面着手。第一，实现表达错误的通用方法，这是错误处理的先决条件。C 语言中的错误往往是通过错误码（error number 或 error code）来表示的，错误码的定义应体现一定的模块性和具体性。

第二，设定报告错误的方法。输出日志记录错误是一种常用的方法，但如何组织输出信息以方便查错很有考究。

第三，制定统一处理错误的原则。这些原则将给软件开发工程师一些评价准则，使得他们在编写错误处理代码时思路更清晰，而不致于随心所欲甚至干脆忽视。各个项目对于错误的处理原则与软件所服务行业的特点是相关的，因此处理原则具有极大的差异性，在本书不打算就这一点展开讨论。

15.1 表达错误的通用方法

在理想情况下，如果一个项目的所有错误码都能事先全部定义好，且工程师也能恰当地使用的话，则根本没有必要花一章的篇幅来介绍错误码的通用定义方法。现实情况是，项目的所有错误码无法在开发之初就事先定义好，而且，要实现各工程师都恰当地使用每一个错误码并非一件易事。

在 C 语言标准库中定义了不少错误码，它们能在系统头文件 errno.h 中找到。但是很遗憾，标准库所使用的方式并不适合运用到现实项目中，因为其采用的策略是尽可能复用错误码，这就造成同一个错误码被不同的函数返回时所表达的意思有可能完全不同。当然，库采用这种方式有它的原因，因为它所提供的功能都是相对单一的，所以出现的错误也不像应用程序那样繁杂。

其实无论采用怎样的错误码定义方法，都需要防范错误码被滥用——重复定义、定义不精确和引用不当。将错误码的定义基于模块将有助于控制错误码的滥用，即使要纠正也更轻松，这里介绍的方法正是基于这一思想的。

15.1.1　错误码格式

错误码是整型数字，一个错误用一个数字表达。错误码数据类型的定义如图 15.1 所示。宏 __error_t_defined 是为了防止 error_t 的定义与 C 库中的定义相冲突。

```
00032: #ifndef __error_t_defined
00033: typedef int error_t;
00034: #define __error_t_defined
00035: #endif
```

图 15.1

错误码的格式如图 15.2 所示。其中最高位（MSB）永远是 1。对于一个有符号的数据来说，最高位为 1 就表示它是一个负值，因此错误码是用负数来表示的。如果一个错误码是 0，它表示成功而非错误。

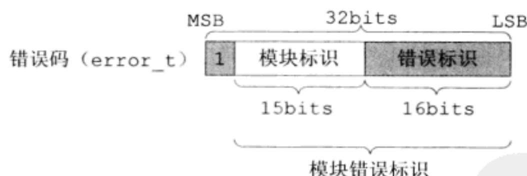


图 15.2

中间的 15 bit/s 是模块标识域，用于指示一个错误属于哪一个模块。由于采用了 15 bit/s 来存放模块标识，因此最多可以为 32768 个模块定义错误码。embedded 项目的模块标识定义于 14.3.1 节中。

最后的 16 bit/s 用于存放错误标识。对于一个模块标识最多可以表示 65536 个错误，对于大多数的模块这一范围足够用了。如果出现一个模块所需的错误码个数大于这个数值，可以考虑通过为之定义多个模块标识的方式进行扩展。

模块标识和错误标识合在一起我们称之为模块错误标识，加上最高位的 1 后，就是一个完整的错误码了。后面将看到，错误码的定义并不是一步到位的，而是先通过定义模块错误标识，

然后在需要时将它转换为错误码。采用这种方式是为了实现上的方便性。

15.1.2 定义方法

我们不应该将整个系统的所有模块错误标识放在一个头文件中进行定义，这样做会造成更改头文件时所有包含它的源文件都得重新编译，结果将造成项目的编译速度下降。

更好的做法是，每一个模块都采用一个独立的头文件来定义它的错误标识。每当一个模块增加、删除或者更改一个错误码时，它只会导致模块本身的源程序需要重新编译，影响范围将大大缩小。

模块错误标识的定义最好不要采用显式指定数值的形式，这种形式并不易于维护。通过运用 C 语言的一些特性，我们可以做到简化错误码的定义和维护。图 15.3 示例说明了 error.h 文件中定义的几个宏，通过使用这些宏将方便模块错误标识的定义和错误码的使用。

```
00037:#define MODULE_BITS       15
00038:#define ERROR_BITS        16
00039:
00040:// if the ERROR_T_SIZE is 4 then the ERROR_BIT should be 0x80000000
00041:// ERROR_BIT is used to make a number as a negative integer
00042:#define ERROR_BIT (1 << (MODULE_BITS + ERROR_BITS))
00043:
00044:#define ERROR_BEGIN(_module_id) ((_module_id) << ERROR_BITS)
00045:
00046:#define ERROR_T(_module_error) (ERROR_BIT | (_module_error))
00047:
00048:#define MODULE_ERROR(_error_t) ((_error_t) & ((1 << ERROR_BITS) - 1))
00049:#define MODULE_ID(_error_t) (((_error_t) & ~(ERROR_BIT)) >> ERROR_BITS)
```

图 15.3

各宏的作用分别是：

- MODULE_BITS（第 37 行）宏定义的是模块标识比特域位数。根据图 15.2 它应是 15 位。
- ERROR_BITS（第 38 行）宏定义了错误标识的比特位数是多少。现在的值是 16。
- ERROR_BIT（第 42 行）宏定义的是 error_t 的最高位为 1 的位。
- ERROR_BEGIN 宏（第 44 行）用于为模块指定第一个错误标识值。
- ERROR_T 宏（第 46 行）则用于最终生成一个完整的错误码。
- MODULE_ERROR 宏和 MODULE_ID 宏用于从一个错误码中得到其所对应的模块错误标识和模块标识。

为了避免显式使用数字来定义错误码，我们可以使用 C 语言中的联合体。图 15.4 示例说明了如何通过使用它和 ERROR_BEGIN 宏定义模块错误标识。

```
00029: #include "module.h"
00030:
```

```
00031: enum {
00032:     // for module management
00033:     ERROR_MODULE_REG_INVMODULE = ERROR_BEGIN (MODULE_MODULE),
00034:     ERROR_MODULE_REG_INVLEVEL,
00035:     ERROR_MODULE_REG_INVCB,
00036:     ERROR_MODULE_REGISTERED,
00037:     ERROR_MODULE_INIT_FAILURE,
00038:     ERROR_MODULE_UP_FAILURE,
00039:     ERROR_MODULE_DOWN_FAILURE
00040: };
```

图 15.4

从图中可以看出，模块的第一个错误标识需要使用 ERROR_BEGIN 宏加以指定，然后通过不指定值的方式定义其他模块错误标识。错误标识名称最好采用“ERROR_模块名_动作_错误描述”这种命名形式，这有利于一眼识别出错误具体所指。由于模块错误标识的定义使用了 C 语言中的联合体，因此增减模块错误标识都很简单。在 MODULE_MODULE 的值为 0 的情形下，图 15.5 示例说明了图 15.4 中所定义的模块错误标识的具体值。注意，各错误标识的最高位为 0 而不是 1。

到这里有读者可能会有疑问：为什么在定义时不直接把最高位置为 1 呢，而是需要最终通过宏 ERROR_T 来转换成真正的错误码？我们知道枚举中的值是依次递增的，也就是说，为了保证错误码值为负数这一前提，我们就需要将宏 ERROR_BEGIN 定义成每个模块中的最小错误码值，这个实现单纯从这点来说也未尝不可。但是，当我们对错误码值用数组管理（后面会涉及）时就不得不通过一些运算，而不能像现在的实现一样直接通过抽取错误标识中的不同位域高效获得。

0	0x0	0x0000	ERROR_MODULE_REG_INVMODULE
0	0x0	0x0001	ERROR_MODULE_REG_INVLEVEL
0	0x0	0x0002	ERROR_MODULE_REG_INVCB
0	0x0	0x0003	ERROR_MODULE_REGISTERED
0	0x0	0x0004	ERROR_MODULE_INIT_FAILURE
0	0x0	0x0005	ERROR_MODULE_UP_FAILURE
0	0x0	0x0006	ERROR_MODULE_DOWN_FAILURE

图 15.5

15.1.3　使用示例

图 15.6 是使用错误码的一个代码片段。对于这个代码片段，我们需要关注以下几点。

- 函数的返回类型是 error_t。
- 返回 0 表示成功。
- 如果需要返回一个错误码，需要使用 ERROR_T 宏将模块错误标识转换为错误码。

```
00044: error_t module_register (const char _name [], module_t _module,
00045:     init_level_t _level, module_callback_t _callback)
00046: {
00047:     module_init_t *p_module;
00048:
00049:     if (_module > MODULE_LAST) {
00050:         return ERROR_T (ERROR_MODULE_REG_INVMODULE);
00051:     }
00052:     if (_level > LEVEL_LAST) {
00053:         return ERROR_T (ERROR_MODULE_REG_INVLEVEL);
00054:     }
00055:     if (null == _callback) {
00056:         return ERROR_T (ERROR_MODULE_REG_INVCB);
00057:     }
00058:
00059:     p_module = &g_modules [_module];
00060:     if (p_module->is_registered_) {
00061:         return ERROR_T (ERROR_MODULE_REGISTERED);
00062:     }
00063:     p_module->p_name_ = _name;
00064:     p_module->callback_ = _callback;
00065:     p_module->is_registered_ = true;
00066:     dll_push_tail (&g_levels [_level], &p_module->node_);
00067:
00068:     return 0;
00069: }
```

图 15.6

15.1.4　提高可使用性

由于现有的模块错误标识没有采用指定固定值的方式，会造成一个错误码的值（以下简称错误码值）因为联合体中内容位置的变动而发生改变这种情形。为此，如果直接将错误码值输出到日志中，就很容易出现同一个错误码值随不同的软件版本却表示不同的错误。另外，如果在日志文件中直接使用错误码值，其可读性也不好。由于一个错误码由三个位域组成，在获得错误码值以后还得根据格式进行解码以了解具体的错误含义。这一切表明，我们需要一种更具可使用性的方法。

毫无疑问，当出现错误并需要将它记录到日志文件中时，错误码的名称（以下简称错误码名）比错误码值更具可读性。在 C 库中提供了一个 errno 变量[①]用于得到出错时的错误码值，以错误码值为参数调用 strerror()函数可以获得错误码名。我们也可以借鉴这一方法。

从程序实现的角度来看，我们必须在代码内保存用于实现值与名转换的字符串数组。最常用的实现方法是在代码中增加一个字符串数组，且需要将之与错误码的定义不断地进行手工同步，这种做法的可维护性并不好且容易出错。接下来要讨论一种自动生成的好方法。

---

① 在有些库的实现中，errno 其实是一个指向任务变量的宏，但这并不影响我们对 errno 用处的理解。

15.1.4.1　名称管理

在说明错误码值与错误码名映射数组如何自动生成之前，需要先解决错误码名的管理问题。我们约定错误标识域和模块标识域的值都从 0 开始，这一约定有利于我们采用数组加以管理。

用于管理错误码名的 errstr_t 数据结构的定义如图 15.7 所示。对于每一个模块，都有一个 errstr_t 类型的变量与之对应，所以图中 g_errstr_array 变量数组的元素个数由整个系统的模块数决定，这可以使用 MODULE_COUNT 加以指定。

```
00030: static struct errstr_t {
00031:     int available_;
00032:     int last_error_;
00033:     const char **error_array_;
00034: } g_errstr_array [MODULE_COUNT];
00035:
00036: #include "errstr.def"
```

图 15.7

errstr_t 数据结构中各变量的含义如下：

- available_变量用于表示所对应模块是否存在名称信息，当存在时用 1 表示。
- last_error_变量则表示所对应模块的最后一个错误标识值是多少，在名称查找时要用到该值。
- error_array_变量是指向模块所有错误码名数组的指针。

读者要注意一下 error.c 中第 36 行所包含的 errstr.def 文件，这个文件是通过自动生成而获得的，这是下面需要涉及的内容。

15.1.4.2　自动生成名称数组

错误码值与名之间的映射关系在有了 errstr_t 数据结构后就可以考虑通过自动生成的形式创建了。工具可以用任何脚本语言或我们熟悉的 C/C++语言。在此，作者采用的是 C++语言，工具命名为 err2str，它的源代码可以从光盘 embedded/code/tools/ err2str 目录中找到。原理很简单：通过对每个模块错误标识定义文件进行字符查找，将其转换为错误码名并加入数组中。

对于模块管理所定义在 errmod.h 内的模块错误标识，通过运行图 15.8 所示的命令可以生成映射数组。所生成的 errstr.def 文件中的内容列于图 15.9 中。

```
$ g++ err2str.cpp -o err2str.exe
$ ./err2str.exe ../../platform/common/inc/errmod.h > errstr.def
```

图 15.8

```
errstr.def
00001: #include "errmod.h"
00002:
00003: static const char *g_errstr_MODULE_MODULE[
         MODULE_ERROR(ERROR_MODULE_DOWN_FAILURE) + 1];
00004:
00005: void errstr_MODULE_MODULE_init ()
00006: {
00007:     g_errstr_MODULE_MODULE[MODULE_ERROR(ERROR_MODULE_REG_INVMODULE)] =
               "ERROR_MODULE_REG_INVMODULE";
00008:     g_errstr_MODULE_MODULE[MODULE_ERROR(ERROR_MODULE_REG_INVLEVEL)] =
               "ERROR_MODULE_REG_INVLEVEL";
00009:     g_errstr_MODULE_MODULE[MODULE_ERROR(ERROR_MODULE_REG_INVCB)] =
               "ERROR_MODULE_REG_INVCB";
00010:     g_errstr_MODULE_MODULE[MODULE_ERROR(ERROR_MODULE_REGISTERED)] =
               "ERROR_MODULE_REGISTERED";
00011:     g_errstr_MODULE_MODULE[MODULE_ERROR(ERROR_MODULE_INIT_FAILURE)] =
               "ERROR_MODULE_INIT_FAILURE";
00012:     g_errstr_MODULE_MODULE[MODULE_ERROR(ERROR_MODULE_UP_FAILURE)] =
               "ERROR_MODULE_UP_FAILURE";
00013:     g_errstr_MODULE_MODULE[MODULE_ERROR(ERROR_MODULE_DOWN_FAILURE)] =
               "ERROR_MODULE_DOWN_FAILURE";
00014:
00015:     g_errstr_array[MODULE_MODULE].available_ = 1;
00016:     g_errstr_array[MODULE_MODULE].last_error_ =
               MODULE_ERROR(ERROR_MODULE_DOWN_FAILURE);
00017:     g_errstr_array[MODULE_MODULE].error_array_ = g_errstr_MODULE_MODULE;
00018: }
00019:
00020: static void errstr_init ()
00021: {
00044:     errstr_MODULE_TIMER_init ();
00045:     errstr_MODULE_MODULE_init ();
00046: }
```

图 15.9

err2str 可以以多个模块错误定义文件为参数，如图 15.10 所示。所生成的文件内容读者可以自行查看。

```
/embedded/code/tools/err2str
$ ./err2str.exe ../../platform/common/inc/errmod.h \
  ../../platform/timer/common/inc/errtmr.h > errstr.def
```

图 15.10

15.1.4.3　实现名称查找函数

我们引入 errstr()函数实现将错误码值转换为错误码名，其实现如图 15.11 所示。

```
00038: const char *errstr (error_t _error)
00039: {
00040:     static bool initialized = false;
00041:     module_t module_id = MODULE_ID (_error);
00042:     int error_id = MODULE_ERROR (_error);
00043:
00044:     if (0 == initialized) {
```

```
00045:         errstr_init ();
00046:         initialized = true;
00047:     }
00048:
00049:     if (0 == _error) {
00050:         return "SUCCESS";
00051:     }
00052:
00053:     if (_error > 0) {
00054:         return "ERROR_ERRSTR_NOT_NEGATIVE";
00055:     }
00056:
00057:     if (module_id > MODULE_LAST) {
00058:         return "ERROR_ERRSTR_INVALID_MODULEID";
00059:     }
00060:
00061:     if (!g_errstr_array [module_id].available_) {
00062:         return "ERROR_ERRSTR_NOT_AVAILABLE";
00063:     }
00064:
00065:     if (error_id > g_errstr_array [module_id].last_error_) {
00066:         return "ERROR_ERRSTR_OUT_OF_LAST";
00067:     }
00068:
00069:     if (0 == g_errstr_array [module_id].error_array_ [error_id]) {
00070:         return "ERROR_ERRSTR_NOT_DEFINED";
00071:     }
00072:
00073:     return g_errstr_array [module_id].error_array_ [error_id];
00074: }
```

图 15.11

第 41 和 42 行分别获得错误码所属模块的标识值和错误标识值，这是第一次使用到 MODULE_ID()和 ERROR_ID()宏。

第 44～47 行调用 errstr.def 文件中生成的 errstr_init()函数，以实现将各模块的错码码名称数组“挂接”到 g_errstr_array 数组上。这里通过隐式初始化的方式（参见 27.1.11 节）省去了在其他地方调用 errstr_init()函数。

第 49～51 行是处理错误码值为 0 的情形。0 表示成功，所以在第 50 行返回字符串“SUCCESS”。第 53～55 行是检查错误码值是否为正，正值表示是一个无效的错误码。第 57～71 行是其他必要的有效性检查代码。在第 73 行通过返回 g_errstr_array 数组中所存放的字符串结束整个函数的实现。

15.1.4.4　完成无缝整合

至此，所有与错误码相关的程序实现都介绍完了，接着需要将这一方法无缝地集成到开发环境中，以进一步提高可使用性。前面展示了如何手工生成 errstr.def 文件，但在现实的项目中可以通过使用 Makefile 做到一旦模块错误标识定义文件被更改，errstr.def 文件就自动重新生成。这正是“无缝整合”的含义所在。

在后面的质量保证篇中将深入介绍光盘内 embedded 目录下的编译系统是如何构建的，尽管本节的内容依赖于后面的内容，但只要读者阅读并理解了第 3 章就不存在理解困难。

error.c 所在目录的 Makefile 如图 15.12 所示，其中已包含了如何通过使用 Makefile 来自动生成 errstr.def 文件的内容。

```
00001: EXE =
00002: LIB = libcommon.a
00003:
00004: INCLUDE_DIRS = \
00005:     $(ROOT)/code/platform/common/inc \
00006:     $(ROOT)/code/platform/arch/x86/simulator/inc \
00007:     $(ROOT)/code/platform/driver/ctrlc/inc \
00008:     $(ROOT)/code/platform/task/common/inc \
00009:     $(ROOT)/code/platform/task/v3/inc \
00010:     $(ROOT)/code/platform/sync/common/inc \
00011:     $(ROOT)/code/platform/sync/v2/inc \
00012:     $(ROOT)/code/platform/device/inc \
00013:     $(ROOT)/code/platform/memory/common/inc \
00014:     $(ROOT)/code/platform/timer/common/inc \
00015:
00016: # all the error definition files should be put into ERROR_FILES variable
00017: ERROR_FILES = \
00018:     $(ROOT)/code/platform/common/inc/errmod.h \
00019:     $(ROOT)/code/platform/task/common/inc/errtask.h \
00020:     $(ROOT)/code/platform/sync/common/inc/errsync.h \
00021:     $(ROOT)/code/platform/device/inc/errdev.h \
00022:     $(ROOT)/code/platform/arch/x86/simulator/inc/errclock.h \
00023:     $(ROOT)/code/platform/arch/x86/simulator/inc/errcon.h \
00024:     $(ROOT)/code/platform/driver/ctrlc/inc/errctrlc.h \
00025:     $(ROOT)/code/platform/memory/common/inc/errheap.h \
00026:     $(ROOT)/code/platform/memory/common/inc/errmpool.h \
00027:     $(ROOT)/code/platform/timer/common/inc/errtmr.h \
00028:
00029: LINK_LIBS =
00030:
00031: include $(BUILD)/c.rule
00032:
00033: $(DIR_OBJS)/error.dep: genmark
00034:
00035: # for cleaning the errstr.def and genmark files
00036: RMS += errstr.def genmark
00037:
00038: genmark: $(DIR_TARGET)/err2str.exe $(ERROR_FILES)
00039:     $(DIR_TARGET)/err2str.exe $(ERROR_FILES) > errstr.def
00040:     @touch genmark
```

图 15.12

这个 Makefile 中存在以下几个值得关注的点。

- 第 17 行定义的 ERROR_FILES 变量用于存放所有的模块错误标识定义文件。另外，当一个定义文件被增加到 ERROR_FILES 变量中时，需要将错误码所在的路径放到第 4 行定义的 INCLUDE_DIRS 变量中，以便编译 error.c 文件时编译器能找到相应的头文件。

- 第 33 行增加了 error.dep 文件对 genmark 文件的依赖关系。genmark 文件的作用后面马上会涉及。这一依赖关系是保证为 error.c 文件生成依赖文件之前先生成 errstr.def 文件。
- 第 36 行将 errstr.def 和 genmark 两个文件加入到 RMS 变量中，以便运行“make clean”时将它们删除。
- 第 38～40 行增加了一个构建 genmark 目标的规则。请注意 genmark 不能定义为假目标，因为它是一个真实的文件，这个文件的作用是指示 errstr.def 文件已经生成了。从这一新增规则可以看出，genmark 依赖于 err2str.exe 和所有的错误码定义文件。也就是说，只要 err2str.exe 或者错误码定义文件发生了变化，就需要重新构建 genmark 目标。这一规则在第 39 行采用 err2str 工具生成 errstr.def 文件，在第 40 行使用 touch 命令生成 genmark 文件。图 15.13 示例说明了规则在依赖关系树中的位置。

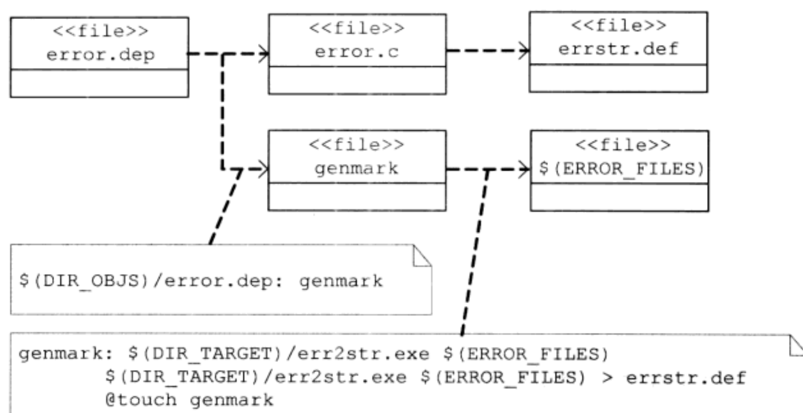


图 15.13

这里需要进一步解释为什么在生成 errstr.def 文件后又要生成 genmark 文件。当 make 正在调用 err2str 生成 errstr.def 文件时，如果我们在终端上使用“Ctrl+C”组合键终止了这一动作，则会得到一个并不完整的 errstr.def 文件。由于这个不完整文件的时间戳已经是最新的了，下一次如果不进行“make clean”的话，make 不会为我们重新生成一个完整的。增加 genmark 文件后就解决了这一问题。只要 genmark 没有生成就意味着 errstr.def 没有生成或不完整，make 就一定会在下一次构建时使用 38～40 行定义的规则重新生成它。

此外，还得保证在生成 errstr.def 之前 err2str 工具已经被编译出来了，这可以通过将编译 err2str 的构建时机放在编译 libcommon.a 之前来实现，如图 15.14 的第 5 和 6 行所示。

```
00001: ROOT = $(realpath ..)
00002: BUILD = $(realpath .)
00003:
00004: COMMON_DIRS = \
```

```
00005:      $(ROOT)/code/tools/err2str \
00006:      $(ROOT)/code/platform/common/src \
00007:      $(ROOT)/code/platform/arch/x86/simulator/src \
00008: ......
```

图 15.14

15.1.5 定义和使用错误码的准则

错误码的定义和使用应当遵守以下准则。

- 尽可能让错误码的名称能表述具体的错误。不要定义类似 ERROR_GENERAL 这样的名称，因为看了也不知所云。错误码的名称应当采用“ERROR_模块名_动作_错误描述”这种统一的格式，以使其更具可读性。
- 尽可能不要复用错误码。有些错误可能在同一个模块的多个函数中都会出现，如果复用同一个错误码，在获得它后并不知道它是由哪一个函数造成的，这不利于我们定位问题。如果一种错误无论在哪里出现我们都能清楚地知道它所表达的意思，那么在这种情况下复用错误码是可以考虑的。
- 错误码定义得越多，所需使用的内存空间也会越大，因为程序中需要存放错误码名称字符串。在内存不是瓶颈的情况下，不应考虑内存的节约问题，而应当集中于思考如何有效地表达错误。
- 模块错误标识的定义文件最好采用 errXXX.h 这种统一的文件命名格式。其中 XXX 是模块名或模块的简写名。

15.2 优化错误日志的输出

错误管理一定离不开报告错误以便我们查错，错误信息大都以日志的形式输出。在日志所提供的信息不足以定位问题的情形下，就需要重现错误。如果错误很容易重现，那基本上能快速地解决。但也存在错误因为不易重现而使得我们难以查错的情况。

在日常编程活动中，我们必须抱着问题很难重现的心态去思考应输出哪些日志信息，这有助于出错时第一时间了解错误。由于大量日志的输出不可避免地会影响程序的性能，所以我们需要思考怎样有效地组织错误日志，以尽可能做到既方便查错又不影响程序的性能。

15.2.1 传统方法

传统的日志输出方法是以面向过程的方式。为了说明，需要借助一个简化的话务处理程序片段，如图 15.15 所示。

```
00019: typedef struct
00020: {
00021:     tsi_slot_t slot_;
```

```
00022:     dsp_channel_t channel_;
00023:     char phone_number_ [MAX_PHONE_NUMBER];
00024:     call_leg_t legs_ [MAX_CALL_LEG];
00025: } session_t;
00026:
00027: typedef struct
00028: {
00029:     session_t session_;
00030: } msg_t;
00031:
00032: enum
00033: {
00034:     ERROR_CALLPROCESSING_INVNUM = ERROR_BEGIN (MODULE_CALLPROCESSING),
00035:     ERROR_CALLPROCESSING_NOSLOT,
00036:     ERROR_CALLPROCESSING_NOCHANNEL,
00037: };
00038:
00039: void thread_call_process ()
00040: {
00041:     msg_t *p_msg;
00042:     while (1) {
00043:         if (0 != msg_receive (&p_msg)) {
00044:             log_fatal ("thread_process_call(): receive msg failed");
00045:             return;
00046:         }
00047:
00048:         session_t *p_session = &p_msg->session_;
00049:
00050:         if (0 != phone_numer_analyze (p_session->phone_number_)) {
00051:             continue;
00052:         }
00053:
00054:         if (0 != slot_alloc (&p_session->slot_)) {
00055:             continue;
00056:         }
00057:
00058:         if (0 != channel_alloc (&p_session->channel_)) {
00059:             continue;
00060:         }
00061:
00062:         if (0 != call_establish (p_session)) {
00063:             continue;
00064:         }
00065:     }
00066: }
00067:
00068: error_t phone_numer_analyze (char _phone_number[MAX_PHONE_NUMBER])
00069: {
00070:     log_debug ("phone number is %s\n", num2str (_phone_number));
00071:     // logical code is put here ...
00072:     if (number is invalid) {
00073:         log_error ("invalid number %s", num2str (_phone_number));
00074:         return ERROR_T (ERROR_CALLPROCESSING_INVNUM);
00075:     }
00076:
00077:     return 0;
00078: }
00079:
00080: error_t slot_alloc (tsi_slot_t *_slot)
00081: {
```

```
00082:     *_slot = INVALID_SLOT;
00083:     tsi_slot_t slot;
00084:
00085:     // logical code is put here ...
00086:     if (there is no resource available) {
00087:         log_error ("no slot is available");
00088:         return ERROR_T (ERROR_CALLPROCESSING_NOSLOT);
00089:     }
00090:
00091:     *_slot = slot;
00092:     log_debug ("slot is %s\n", slot);
00093:
00094:     return 0;
00095: }
00096:
00097: error_t channel_alloc (dsp_channel_t *_channel)
00098: {
00099:     *_channel = INVALID_CHANNEL;
00100:     dsp_channel_t channel;
00101:
00102:     // logical code is put here ...
00103:     if (there is no resource available) {
00104:         log_error ("no channel is available");
00105:         return ERROR_T (ERROR_CALLPROCESSING_NOCHANNEL);
00106:     }
00107:
00108:     *_channel = channel;
00109:     log_debug ("channel is %s\n", channel);
00110:
00111:     return 0;
00112: }
```

图 15.15

thread_call_process()是一个线程的入口函数，负责处理接收到的话务消息。

这个程序片段使用了三个级别的日志：fatal（致命）、error（错误）和 debug（调试），严重程度分别从高到低。带有日志功能的程序，通常能通过某种形式控制允许输出的等级。比如，如果只允许输出到错误级的话，那么低于它的调试级的日志就不能输出。显然，输出日志的级别越低，其所输出的信息就越细。大多软件在正常运行时并不输出调试级信息，这是考虑系统性能而做出的选择。

对于图 15.15 中的程序，当 phone_number_analyze()、slot_alloc()或 channel_alloc()中出现错误时都会输出错误日志。这一简化程序中输出日志的方式，作者称之为面向过程的平面化输出。这种方式具有以下特点：

- 日志的输出动作是分散在各函数中的，所获得的日志很像“流水账”，查错时不容易理出整体逻辑。
- 由于错误日志是分散的，当需要增减日志时不易维护。
- 输出的日志信息很容易出现重复而造成大量的冗余。
- 这种形式的错误处理能通过日志反映出程序的运行路径，这对于分析问题有一定的帮助。

采用这种方式输出的错误日志，由于每一个函数看到的只是其函数内部本身有限的信息，因此更具局限性。为了更有效地组织错误信息，我们需要探索其他的方法。

15.2.2 更有效的方法

为了解决面向过程方式所存在的问题，我们可以考虑将出错时的输出日志的动作放在处理逻辑的更上层。对于图 15.15 中的程序，处理逻辑的更上层是在 thread_call_process()函数内，因为不论后面要调用多少个函数，消息都是在这个函数中收到并进行处理的。另外，thread_call_process()所调用的所有函数，都是基于 session_t 结构进行处理的。如果将出错处理放到 thread_call_process()函数中，修订后的代码如图 15.16 所示。

```
00019: ......
00038:
00039: void thread_call_process ()
00040: {
00041:     msg_t *p_msg;
00042:     while (1) {
00043:         if (0 != msg_receive (&p_msg)) {
00044:             log_fatal ("thread_process_call (): receive msg failed");
00045:             return;
00046:         }
00047:
00048:         session_t *p_session = &p_msg->session_;
00049:
00050:         error_t error = phone_numer_analyze (p_session->phone_number_);
00051:         if (0 != error) {
00052:             goto error;
00053:         }
00054:
00055:         error = slot_alloc (&p_session->slot_);
00056:         if (0 != error) {
00057:             goto error;
00058:         }
00059:
00060:         error = channel_alloc (&p_session->channel_);
00061:         if (0 != error) {
00062:             goto error;
00063:         }
00064:
00065:         error = call_establish (p_session);
00066:         if (0 != error) {
00067:             goto error;
00068:         }
00069:
00070:         continue;
00071: error:
00072:         log_error ("error is ", errstr (error));
00073:         log_error ("phone number is %s\n", num2str (p_session->phone_number_));
00074:         log_error ("slot is %s\n", p_session->slot_);
00075:         log_error ("channel is %s\n", p_session->channel_);
00076:     }
00077: }
00078:
00079: error_t phone_numer_analyze (char _phone_number[MAX_PHONE_NUMBER])
00080: {
```

```
00081:     // logical code is put here ...
00082:     if (number is invalid) {
00083:         return ERROR_T (ERROR_CALLPROCESSING_INVNUM);
00084:     }
00085:
00086:     return 0;
00087: }
00088:
00089: error_t slot_alloc (tsi_slot_t *_slot)
00090: {
00091:     *_slot = INVALID_SLOT;
00092:     tsi_slot_t slot;
00093:
00094:     // logical code is put here ...
00095:     if (there is no resource available) {
00096:         return ERROR_T (ERROR_CALLPROCESSING_NOSLOT);
00097:     }
00098:
00099:     *_slot = slot;
00100:     return 0;
00101: }
00102:
00103: error_t channel_alloc (dsp_channel_t *_channel)
00104: {
00105:     *_channel = INVALID_CHANNEL;
00106:     dsp_channel_t channel;
00107:
00108:     // logical code is put here ...
00109:     if (there is no resource available) {
00110:         return ERROR_T (ERROR_CALLPROCESSING_NOCHANNEL);
00111:     }
00112:
00113:     *_channel = channel;
00114:     return 0;
00115: }
```

图 15.16

新实现的最大变化在于，错误日志的输出全部放在 thread_call_process()函数中（第 72～75 行），而其他的被调用函数中不包含任何的日志输出行为。另外，在第 72 行将具体的错误码名打印出来，错误码名可以告诉我们是什么样的错误发生了且发生在哪一个模块。

采用这种错误处理方式的依据是，不论是什么类型的错误都与相关的业务逻辑有关。也就是说，错误信息是面向业务逻辑进行集中输出的，输出的日志不再是一个个小片段。

在这个示例程序中只输出了业务逻辑中 session_t 数据结构相关的信息，在实际项目中还可根据情况输出其他的信息以便查错。比如，如果 msg_t 数据结构还存在其他的信息，且出错时了解这些信息有助于分析错误成因的话，也可以选择在出错时将它们输出。由于出错时的日志输出是集中管理的，添加、删除将变得容易，也不容易出现冗余日志，更易于引导工程师（尤其是后续的维护工程师）更全面地考虑哪些日志需要输出以方便查错。

面向业务逻辑的集中输出方式也存在一些弊病。其一，由于所有被集中控制点调用的函数只是返回一个错误码，其无法表达错误出现时函数的调用路径，而调用路径可能对于出错成因分析至关重要。在这种方式下，为了获得程序的调用路径，就需要考虑设计一种机制或引入一

个变量用于记录程序的运行路径，并在出错时从集中控制点一并输出。

其二，程序在处理逻辑的过程中一定会产生一些中间信息，这些信息有可能并不是直接放在集中控制点所能看到的数据结构中的，而这些中间信息对于出错成因分析有可能相当重要，此时集中输出方式就表现得有些无能为力了。为了解决这个问题，可以采用两种做法。第一种是更改集中控制点所能看到的数据结构，将中间结果也存入其中，然后在集中控制点处将其一并输出；第二种则是在每个函数内部将中间结果直接输出。第一种做法会造成数据结构更加复杂和难以维护而导致结构退化；而第二种做法其实是结合面向过程的平面化输出和面向业务逻辑的集中输出这两种方式。作者更加推崇后者。

不论是面向过程的平面化输出还是面向业务逻辑的集中输出，每种方式都有一定的优缺点，所以我们需要有机地结合两种方式。在实践中，应考虑以面向业务逻辑的集中输出方式为主，以面向过程的平面化输出方式为辅。

15.3　平台和框架层的错误处理

错误处理是一个全局性问题，平台和框架的开发（参见第 17 章）也毫无例外地应当考虑错误处理。

平台在设计时就应当考虑提供检查各类资源使用状况的手段。平台通常会做大量的资源封装，而资源泄漏是嵌入式软件开发中非常常见的问题。如果平台提供检查资源使用情况的手段，则有助于发现泄漏问题，帮助查错。

平台不应当在其内部输出过多的错误日志。除非一种错误的出现会造成系统无法继续运行，否则平台应当通过返回错误码的形式告诉应用层，对出现的错误如何处理应当由应用层去考虑，因为应用层位于更高层次，能更好地组织错误输出信息。

至于框架层，完全可以采用与应用层相类似的手段去考虑错误处理。框架的错误处理通常相对容易考虑，因为框架本身就是针对一定的应用模式的。

15.4　小结

错误管理应是软件产品不可或缺的用户需求。完备的错误管理功能有助于提高程序的可使用性和可查错性。错误管理设计需要在软件设计之初将它与其他的用户需求功能一样并重考虑。错误管理的缺失往往会造成在软件投入使用后需要大量的排错“消防员”。错误管理的首要条件是，在系统中制定有效的错误表达方法。

传统的面向过程的平面化输出看到的只是数据结构的片段，难以做到有效地描述出错时的现场；面向业务逻辑的集中输出，错误日志不再是片段。各种方式各有利弊，我们需根据需要结合使用。

第 16 章 目录结构管理，使项目进展更顺利

在众多影响软件质量的细节中，项目的目录结构是一个容易被轻视的话题。

16.1 规划目录结构的意义

项目目录结构对于软件开发活动的作用不可小视，表面上它只是为目录取名和定夺各目录存放位置这么简单，但更深层的作用作者认为有三。

16.1.1 书架功能

图书馆如果没有书架，那么所有的书都难以有序地存放，就一定会出现混乱的局面，借一本书的时间成本就必然提高。软件项目中的每一个源程序文件如同图书馆中的一本书，如果随便地放入没有经过精心规划的目录中，那如同图书馆中没有书架一样，当需要检索一个文件时存取成本将更高，或者需要很好的记忆力以辅助查找。

良好的目录结构应能反映出程序的层次感和模块化。通过这种方式组织项目文件，在需要查找文件时有助于快速地收敛并缩小查找范围。另外，当增加一个文件时，如果存在有序的目录结构，我们也能毫不费力地决定新文件应当放在什么地方，而不会让我们在做这一决定时感到痛苦。

16.1.2 意识引导

在一条到处都是垃圾的街道上，行人在需要丢垃圾时很有可能会心安理得地“出手”，而如果街道总是被打扫得干干净净，行人就会不好意思这样做，这就是有名的“破窗理论”。同样，项目目录结构管理也存在类似的环境暗示性和诱导性。

在一个目录组织得井井有条的项目中，我们在创建一个新文件时往往会主动地思考应当将它放到哪一个目录中更合适，如果没有合适的目录用于存放新文件也会很自然地思考是否应新

增一个子目录。反之，一个不讲究目录结构的项目，最终结果就是大家都乱放。

更为深远的影响是，一个有序的目录结构往往会暗示这个项目的要求是很高的，“连目录结构都如此讲究，更不用说在设计和编码方面的要求了”，这种暗示将带来一种积极的心态。

16.1.3　加速新手上手

当新成员加入开发团队时，好的项目目录结构有助于加快他们的上手速度。一个良好的项目目录结构往往会体现出软件的模块性，那么新手很容易地能将目录与程序实现对应起来。在这种情形下，当他被要求要熟悉某一模块时，只要告诉他“你只要看某某目录中的代码就行了”，而不是说“这一模块的文件有的在 A 目录中，有的在 B 目录中，你需要边看代码边了解它们的具体位置”。

熟悉一个目录结构混乱的项目是一件很痛苦的事情，为了防止出现这种痛苦，最好在项目的初期对目录结构进行规划。

16.2　出色目录结构的特点

一个好的目录结构应当考虑以下几个方面。

- 体现功能性。软件项目通常需要包含除了源程序以外的其他文件，比如帮助文件、编译出来的库文件和可执行文件，以及必要的参考文档等。目录结构的设计应当凸显这种功能性。
- 折射软件的层次感。在 14.1 节中指出了软件模块的分层概念，好的目录结构应当体现平台层、框架层和应用层。
- 表达软件的模块化。目录结构应当体现模块化，这是目录结构管理最基本的要求之一。

16.3　一个示例

图 16.1 是一个经过精心规划的项目目录结构。这正是本书的 embedded 和 ClearRTOS 项目所采用的目录组织形式。

project 目录是整个项目的根目录，在它的下一层包含 code、docs 和 build 三个子目录，分别用于存放项目源程序、文档和编译出来的库和可执行文件，很好地体现了功能性。

code 目录下又包含了三个目录，即 platform、framework 和 application，很好地折射了软件的层次感。在这三个目录的下面将分别存放归属于它的软件模块。

在 platform、framework 和 application 目录下存放的就是各个模块，以表达软件的模块化。具体说来，在 framework 目录下，存在一个 fsm 目录用于存放状态机框架的源代码。每个模块

的目录下又有 inc 和 src 两个目录，分别用于存放头文件和源文件。在 src 目录下，还将包含 robjs 和 dobjs 两个在编译过程中自动生成的目录，用于存放编译时生成的目标文件和依赖关系文件，这又体现了功能性。

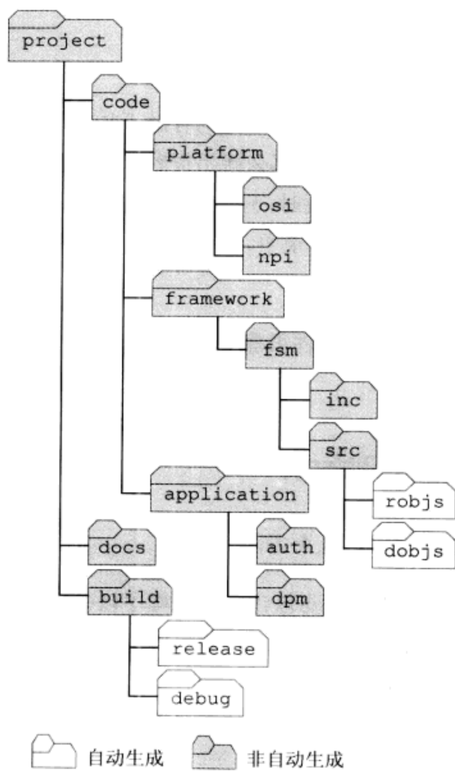


图 16.1

在 build 目录中存在两个同样是在编译期间自动生成的 release 和 debug 目录，分别用于存放发布版和调试板的库及可执行文件。

16.4　小结

项目目录结构管理的重要性体现了软件开发无小事这一思想。好的目录结构能潜移默化地影响工程师的行为，并帮助提高他们的学习和工作效率。

通过将功能性、层次感和模块化反映在项目的目录结构中，有助于项目的顺利推进。

第 17 章 平台与框架开发，高质量软件打造之路

平台与框架开发可以说是软件行业的热门。在解释为什么说平台与框架开发是高质量软件打造之路之前，我们需要先了解系统库、平台和框架三者的区别。

17.1 区分系统库、平台和框架

系统库、平台和框架这三者的区别可以从抽象层次上进行审视，抽象层次从低到高依次是系统库、平台和框架。图 17.1 示例说明了这三者间的关系。在软件行业中，经常会用到“上层软件”这样的词，其中的“上”正是指抽象层次的高低。

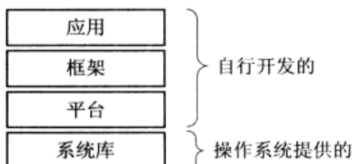


图 17.1

抽象的角度有两种，一种是代码实现的角度，另一种是业务逻辑的角度。图 17.1 所表达的抽象层次是从代码实现的角度看的。越处于上层对代码实现的抽象层次就越高，但却越接近业务逻辑细节，即从业务逻辑角度来看更具体（而非更抽象）。

17.1.1 系统库

系统库（system library）大家都很熟悉。比如我们开发时常用到的 C 语言库，它包含调用操作系统功能的接口函数和其他用于简化应用程序实现的公共函数。系统库并不了解上层应用软件的业务模型，相反，它只能被调用以实现特定的功能。系统库中的函数可以比作各种规格的“螺帽”。

17.1.2　平台

平台（platform）有着比系统库更高层次的抽象。平台可以包含使用系统库中的函数实现更加复杂的功能模块，也可以有为上层软件提供的通用软件模块（如链表）。比如，在平台中可以提供一个日志输出的软件模块，这一模块提供接口函数以便调用者将日志输出到文件、另一台网络主机或其他地方。从上层软件来看，只需调用模块的接口函数，而不用关心日志模块的具体实现。

平台通常会与“跨”字组合成为“跨平台”。此时，“跨”字的使用能很好地表明引入平台的目的。众所周知，无论是桌面软件领域还是嵌入式软件领域都有多种操作系统，在不少情形下我们希望被开发的应用软件能在多个不同的操作系统上运行。然而，不同种类的操作系统所提供的系统库并非完全一样，如果将这种不一样直接反映到上层应用软件的实现中，将造成应用软件开发的复杂度和工作量成倍地增加，通过开发平台这种方式就能很好地解决这一问题。

平台也可能存在对系统库中某些函数的简单封装。比如，提供一个 osal_snprintf()函数用于实现对系统库中 snprintf()函数的封装。对于这一简单的封装请不要认为它是多余的，一个平台如果在设计时并没有向上层提供应用所需的完整接口函数集的话，很有可能造成下次做跨操作系统移植时将花费更大的努力，因为不完整的函数集一定会造成应用层直接调用操作系统中的库函数。平台可以理解为提供比“螺帽”更大的“标准件”。

从某一应用程序的角度来看，只需使用有限的系统库的功能，如果将这些有限的功能进行封装并作为平台的一部分，然后向上层提供封装好的函数接口，平台之上的软件就可以通过调用它们来实现业务逻辑。当一个系统希望从一个操作系统移植到另一个操作系统上时，只需将平台中的封装函数针对新的操作系统重新实现一遍就好了，并保持向上的函数接口不变。平台的引入，能将跨操作系统的移植工作限定在一个小范围内，并达到使其之上的软件不需要做任何更改的目的。

17.1.3　框架

框架（framework）是针对特定的应用所设计出来的更高抽象层次的软件模块，它对应用领域（domain）的概念进行封装和抽象。比如，状态机框架是针对状态机实现的，不论是什么应用，只要采用了状态机，所需的元素就都是一样的，存在事件、状态迁移等概念，这些内容完全可以纳入到框架中去实现，而之上的应用层只要关心定义状态和状态间的迁移，以及实现对各事件的处理即可。

再比如，只要使用 TCP 进行网络通信，就一定存在套接字打开、端口绑定、侦听和连接等操作。这些操作可以通过设计框架的方式对上层软件进行屏蔽，以至从应用层来看，如果要开始网络通信，只需要创建一个链路就行了，从原来的三四个动作简化成一个。

由此看来，框架为相类似的应用业务提供了一个程序实现骨架，而具体的应用只是在这个

骨架之上填上“皮“和“肉”。框架之于系统库和平台有一个很明显的特点，就是使用系统库和平台时我们得主动调用它们的函数以实现功能，但框架却不同，一旦初始化好了框架以后，我们并不需要主动调用框架的功能，相反，框架会通过回调的形式调用我们所实现的“皮”和“肉”（即各回调函数）。

基于虚拟机的编程语言的出现为平台与框架开发打开了全新的篇章。这种形式的好处是，跨平台开发工作成了开发虚拟机工作的一部分，是由语言的创造者完成的。编程语言良好的跨平台性将显著地促进其上平台与框架的开发。

系统库、平台和框架是一种逻辑概念，其主要体现在抽象层次上。在现实项目中，这三者都是以库文件的形式出现的。尽管从文件的角度来看三者都是库，但我们需要从抽象概念上对之有清晰准确的认识。

17.2　本质和优点

抽象和代码复用是平台与框架开发的本质。采用平台与框架开发将带来如下几个好处。

第一，采用平台与框架开发，可以提高后续项目的开发效率。平台和框架一旦被打造出来，我们就可以很容易地复用它们。

第二，采用平台与框架开发容易通过测试保证代码质量。由于平台与框架更能体现模块性，以及具有更高的抽象层次，这为单元测试（参见第 28 章）和可测试性设计（参见 28.7 节）都提供了便利。

第三，采用平台与框架开发有助于提高缺陷的修复效率和减少缺陷数量。在这方面框架设计尤为突出。当有时因为一些概念不清等原因而造成缺陷时，修复框架中的一个缺陷能立即带来多个模块的缺陷同时被修复。反之，同一类缺陷有可能需要在多个模块中分别进行修复。而且，由于平台和框架的开发具有一定的延续性，这使得我们不容易犯以前曾经犯过的错。

第四，通过平台与框架开发可以更好地积累知识和经验。这一点在个人、项目团队和公司三个方面都成立。各软件公司所服务的行业都有自己的特点，而在这些特点的背后我们总是能找出一定的软件架构模型，这就很适合通过开发平台与框架的方法去沉淀知识和经验。可以说，一个公司的软件开发实力可以从其是否拥有自己的平台和框架去考量。出色的软件公司，往往专注于持续打造与改善自己的软件平台和框架。

第五，通过平台与框架开发有利于简化上层软件的程序实现。平台和框架都是一种比库更高层次的抽象，抽象所带来的好处是能向上层简少接口函数的数量，使上层软件的实现更加简洁，也更易于开发和维护。

第六，平台与框架的打造过程能有效地磨练开发团队的设计能力。同样是设计，但平台与

框架开发对于设计能力的要求更高，这与平台和框架处于更高的抽象层次有关，正是这种高要求有助于提升开发团队的设计能力。

第七，通过平台与框架开发可以帮助提高软件项目的可开发性。在第 18 章专门就可开发性设计进行了探讨，而可开发性的提高就需要运用到（跨）平台开发技术。

第八，通过平台与框架开发有助于我们运用开源工具去保证软件质量。嵌入式软件的规模和复杂度都在持续增长，与此同时，对于软件质量的要求也越来越高。在开发过程中合理地运用工具将有助于提升软件质量。但是，并不是每一个软件项目都有足够的预算去购买商用软件。开源工具的蓬勃发展为项目团队解决资金问题提供了另一种途径。

不论是代码覆盖（参见第 29 章）、静态分析（参见第 30 章）、动态分析（参见第 31 章）还是性能分析（参见第 32 章）都有现成的开源工具，但它们都有一个共性——大都运行在 Linux 操作系统上。要使用这些开源工具去保证嵌入式软件产品的质量，就需要运用到平台开发技术。通过将嵌入式软件项目开发成也能在 Linux 操作系统上运行的方法，使我们能使用开源工具。第 18 章就如何让嵌入式程序实现跨平台提供了一些思路。

虽然平台与框架开发具有如此多的优点，但其也给我们带来了挑战。平台与框架具有更高的抽象层次，它对于软件开发工程师的能力要求也更高，更需要相关人员具备一定的行业经验。平台与框架的开发能力能很好地反映软件公司的实力，或者可以说，软件公司间的同业竞争其实是平台与框架的开发能力之争。

17.3 确立架构模型

说到平台和框架很容易让人想到架构（architecture）。通俗地说，架构就是指结构、组织形式。架构有大有小，大的如系统架构；小的可以是一个数据结构。

前面的图 17.1 示例说明了一个嵌入式系统的架构，它看起来是那样的整齐，从上到下“码”得很好。但现实项目很难做到这样的架构，有可能像图 17.2 那样，更有可能完全没有平台和框架的概念而蜕变成图 17.3 所示的那样。在打造平台和框架时，我们需要先确定整个系统的架构模型。

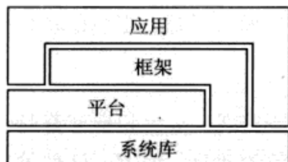


图 17.2

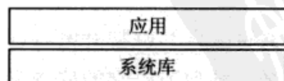


图 17.3

对于图 17.2 所示的架构，让我们看看它所表达的意思。首先，平台是基于系统库的，框架

则是基于平台和系统库的。也就是说，框架并没有完全基于平台，而是有一部分直接调用了系统库中的函数。接下来看一看应用层，应用层使用了平台、框架和系统库的函数。像图 17.2 这样的架构并不是很好，更好一点的架构如图 17.4 所示，即应用只使用平台和框架的功能，而不直接使用库的功能，且框架也只是基于平台的。

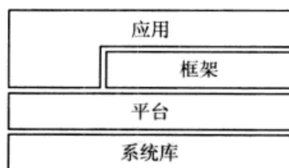


图 17.4

当我们确定了系统的架构模型后，接下来的设计工作就应该按照该模型来展开。当然，这不能一步到位，而是一个演进的过程。

17.4　小结

平台与框架开发技术的运用有助于避免出现“重新造轮子”的现象，它不仅有助于提高产品的质量和缩短产品开发周期，还有利于积累项目经验和知识。

由于平台与框架具有更高的抽象层次，它对于软件开发工程师的能力要求也更高。软件公司之间的竞争其实也包含软件平台和框架的竞争，它甚至是决定性的技术要素。



第 18 章

可开发性设计，一种高效且经济的开发模式

这是一个竞争的时代，除了客户的成熟造成对于嵌入式产品的质量要求越来越高外，还得面对同行的竞争压力。将产品快速地投向市场是任何一个组织所希望的，那如何才能做到呢？总的说来，一要资源，二靠方法。方法对于软件就是软件开发方法。

可开发性设计就是在开发软件产品时，软件的设计应考虑方便工程师开展开发工作，或者说在设计时应考虑如何提高开发效率。

18.1　可开发性问题一瞥

为了了解可开发性问题，让我们看两个比较普遍的现象。第一个现象是，在嵌入式开发中调试设备往往是紧缺资源，不可能做到让工程师人手一台。

第二个现象与传统的嵌入式开发模式有关。图 18.1 示例说明了嵌入式系统的传统开发环境。首先，软件工程师在开发主机上编辑并编译好程序文件，然后，将其传输并保存到嵌入式产品中并通过运行进行验证。如果验证不通过就需要重复上面的动作，这个过程往往比较漫长。

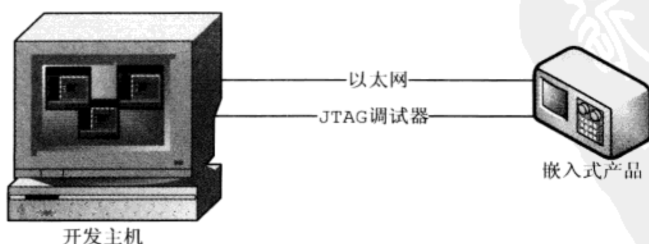


图 18.1

在这种传统开发模式之下，软件工程师将大量的时间花费在交叉编译、文件传输等环节中。值得一提的是，嵌入式系统开发的调试手段也相对落后，其效率远不如在 Windows 操作系统上

采用 Visual Studio 进行桌面软件开发那么方便。

这两个现象造成的最终结果就是开发效率低下，或者说造成了可开发性问题。

18.2 可开发性设计的内涵

为了解决可开发性问题，需要在产品设计之初就将可开发性与产品功能并重考虑，通过运用软件设计这一方法来提高开发效率。解决可开发性问题需要考虑以下几个方面。

- 尽可能让软件的开发和验证不是完全在嵌入式产品上完成。比如，是否可以将一部分程序的开发和验证放到桌面操作系统上完成呢？当在桌面操作系统上开发完成后，再将应用程序放到设备中进行最后的验证，以此来减少对设备的依赖。在验证过程中如果出现了新的问题，则再采用传统的嵌入式开发方法进行调试。
- 采用更高效的开发和调试工具。比如，利用桌面平台的开发工具就能很有效地提高开发效率。除了编译和调试效率外，在桌面环境下调试程序还不需要用到 JTAG 调试器，这可以减少 JTAG 调试器的采购数量，从而节省开发费用。

归根结底，上面所谈及的两个方面都是将嵌入式应用程序的开发调试从一味地依赖嵌入式环境中脱离出来。这也正是嵌入式开发中可开发性设计的核心思想。

18.3 引入设备抽象层

图 18.2 示例说明了一个传统的嵌入式产品开发所涉及的元素。从嵌入式设备的角度来看，嵌入式库是对嵌入式操作系统所提供服务的一种封装，它可能包括 socket 库、C 库或 C++的 STL 库等。应用程序是通过调用库来使用操作系统所提供的服务的。从开发主机的角度来看，软件开发工程师在主机上完成代码的编写，然后通过交叉编译环境完成代码编译。另外，JTAG 调试软件则提供一种采用 JTAG 调试器进行调试的人机界面。

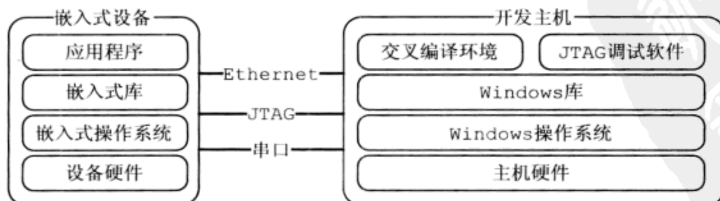


图 18.2[①]

从解决可开发性问题的角度来看，必须打破图 18.2 所示的结构。由于希望嵌入式产品的应

① 图中以 Windows 桌面操作系统为例，实际上也可以是其他的桌面操作系统。

用程序可以在开发主机上直接进行开发和调试，因此嵌入式产品的应用程序不能直接依赖于嵌入式库。由此看来，第一个变化是要去除应用程序对于嵌入式库的直接依赖，即要增加一个设备抽象层，如图 18.3 所示。

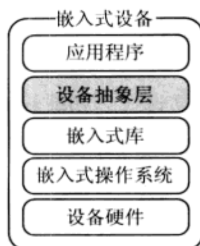


图 18.3

采用这种形式以后，应用程序就直接依赖设备抽象层了。设备抽象层应当理解成一个平台而不是框架，它需要被开发成既可以在嵌入式环境中运行也可以在开发主机中运行。也就是说，解决可开发性问题需要用到上一章介绍的（跨）平台开发技术。

设备抽象层所提供的服务应当尽可能地接近各种操作系统提供的服务，而不应当对操作系统的服务进行扩展，将设备抽象层做得尽可能简单是从开发经济性的角度考虑的。为了描述这种简单性，通常我们会说将设备抽象层做得比较“薄”。反之，如果说“厚”就是指将功能做得比较复杂或抽象层次比较高。

增加了设备抽象层以后，如果软件所需用到的硬件资源在开发主机环境上都有，那么开发工作就可以放在开发主机上进行，这样就得到了如图 18.4 所示的开发环境。

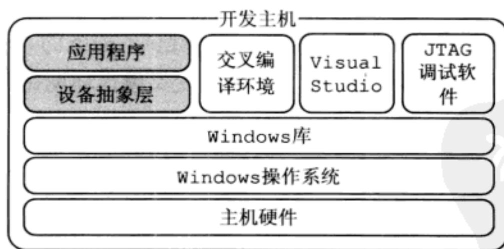


图 18.4

无论嵌入式产品的应用程序多么万变，设备抽象层的功能应该总是相对恒定和简单。当设备抽象层开发完成以后，就可以将精力集中于在开发主机上从事应用程序的开发。

设备抽象层至少需要两个版本，一个运行在嵌入式设备上，另一个则运行于开发主机上。对于嵌入式版本的设备抽象层，仍需要采用传统的嵌入式开发方法进行开发；而开发主机版本的设备抽象层就能完全采用开发主机的开发环境进行开发。

18.4　更复杂的设备抽象层

很多的嵌入式应用程序都需要使用嵌入式设备所特有的硬件资源，比如使用时隙交叉芯片完成 E1 光纤时隙的交换等，在这种情况下开发主机设备抽象层的实现会变得复杂。图 18.5 示例说明了这类嵌入式设备在开发主机上进行开发时的元素，其中最大的变化存在于设备抽象层。

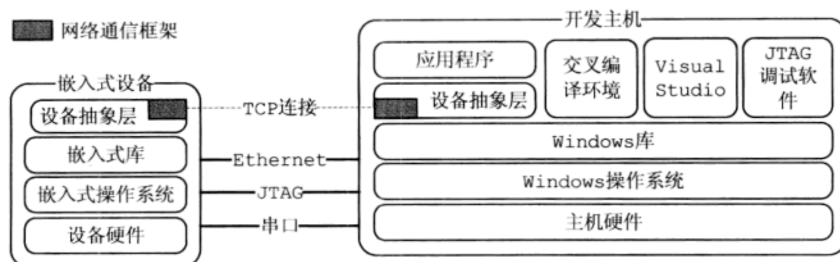


图 18.5

从图中可以看出，此时的设备抽象层被分成了两块，开发主机版本的设备抽象层中的一部分功能需要依赖嵌入式版本的设备抽象层。从图中还可以看到，开发主机版本的设备抽象层和嵌入式版本的设备抽象层通过“TCP 连接”进行互联。

当应用程序在开发主机上开发时，开发主机版本的设备抽象层在收到应用程序的函数调用后，将这一调用通过网络通信框架发送给位于嵌入式版本的设备抽象层，并阻塞应用程序以等待对端的回应。嵌入式版本的设备抽象层收到请求后，调用嵌入式库完成真正的设备访问，然后将访问结果通过网络通信框架发送给位于开发主机中的设备抽象层，返回的结果可能是从设备硬件中获取到的数据，也可能只是一个简单的操作结果。开发主机版本的设备抽象层收到了对端的回应后，将数据或结果返回给应用程序。图 18.6 所示的顺序图示例说明了应用程序完成一次硬件资源访问的消息流。

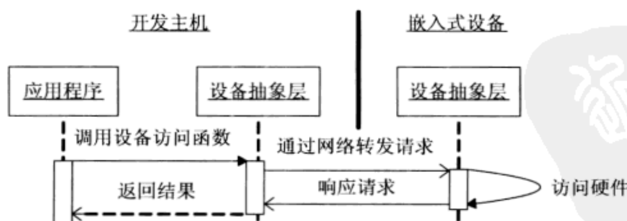


图 18.6

由于嵌入式设备特有硬件资源的存在，造成了程序在开发主机上开发时设备抽象层被一分为二。当然，当应用程序运行在嵌入式产品上时，嵌入式版本的设备抽象层并不需要网络通信框架的存在，而是直接采用函数调用的形式。

18.5 图形界面的可开发性设计

在包含图形界面（UI）的嵌入式产品中，图形界面的开发是整个产品开发活动中的重要内容之一。而图形界面开发的很大一部分工作内容是界面的编辑，传统的做法是需要反复地将开发主机中的界面图片传输到嵌入式设备中，然后通过运行程序来查看效果。

18.5.1 增强设备抽象层

为了提高开发效率，更好的做法是仍然采用前面所提到的引入设备抽象层的方法，将图形界面的开发向开发主机转移。此时，设备抽象层需要增加图形引擎的功能。图 18.7 示例说明了主机开发环境的组成元素。



图 18.7

时下流行的 Android 开发平台就通过提供模拟显示终端的方式很好地解决了图形界面的可开发性问题。在 Windows 操作系统上，开发者可以通过 Andriod 所提供的模拟窗口看到所开发的图形界面在真实手机设备上的效果。

18.5.2 提供可视化编辑环境

为了提高图形界面的开发效率，我们除了可以引入设备抽象层外，还可以提供一个可视化的界面编辑环境。

在开发桌面系统软件时，往往都有一个成熟的可视化开发工具（比如 Windows 上的 Visual Studio），以实现所设计的界面达到所见即所得的效果。但在嵌入式系统开发中就不具备这样的条件了。如果界面开发工作是一个反复、持续的过程，我们就有必要自行开发一个可视化界面编辑工具来简化这项工作。

18.6 其他可开发性设计

引入设备抽象层并不是可开发性设计的全部，在设计中还存在其他需要考虑的因素。比如：

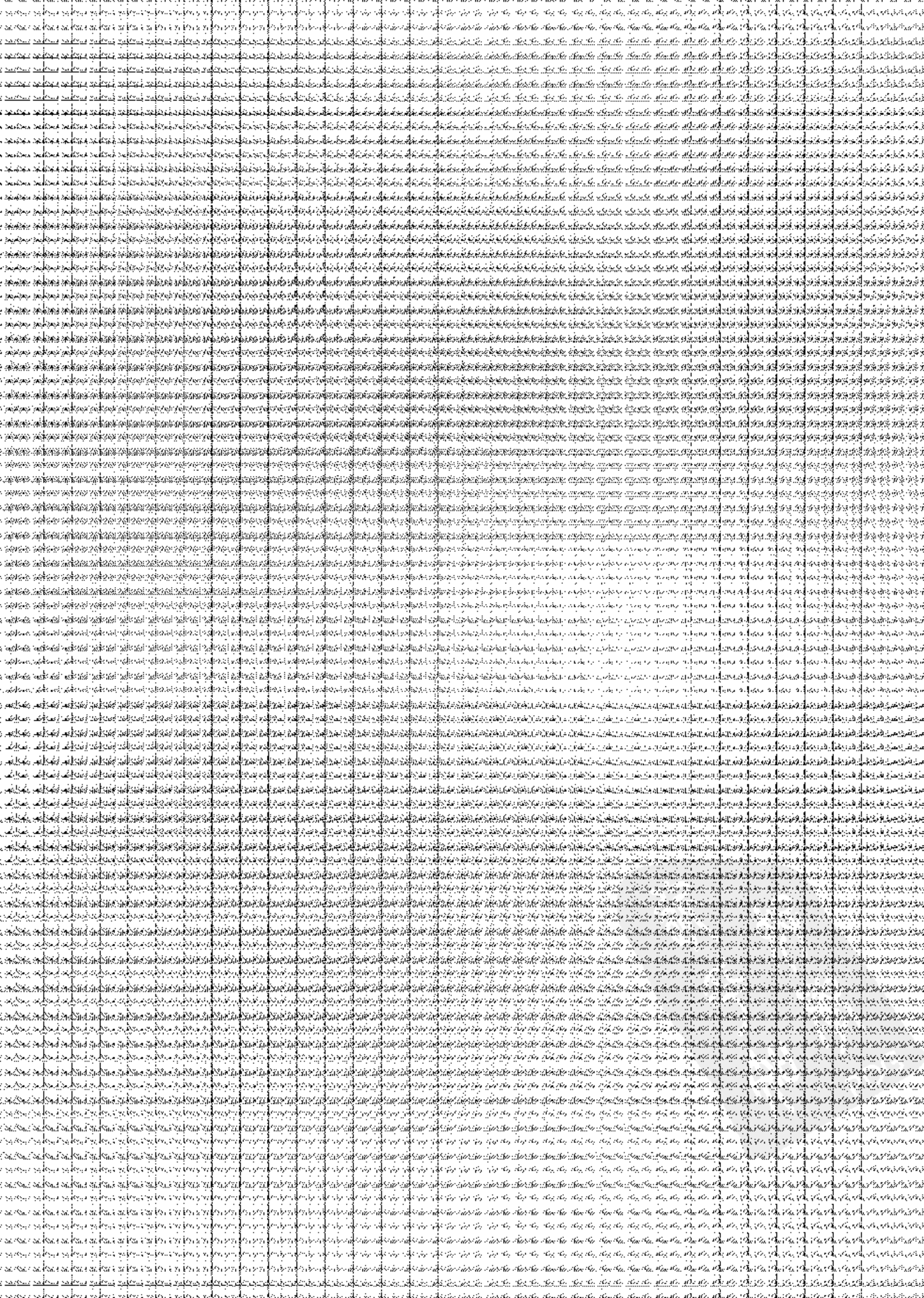
- 当嵌入式产品中的程序采用了多进程的方式时，应当设计成各进程可以被单独重启。为了实现这一目的，要求各进程间能动态建立通信通道。这在一定程度上会增加开发的复杂度，但好处就是能提高开发期间的调试效率。
- 影响调试效率的功能应设计成能被方便地禁用。比如，像硬（软）件狗（watchdog）、进（线）程健康检测（health check）这样的功能，在调试期间如果不禁用，则会导致调试根本无法正常进行。
- 增加开发所使用嵌入式设备的资源数量（或容量）。比如，为开发所使用的嵌入式设备提供更大的内存，避免因为内存资源有限而限制调试方法。

可开发性设计的重要性只有在项目存在可开发性问题时才能让人意识到，而且其影响与项目的规模、复杂度、开发周期等因素有关。具备可开发性设计的意识有助于我们思考如何通过设计来提高开发效率和降低开发成本。

18.7　小结

要实现产品的高效开发，不能只关注项目资源的配置，更要在开发和设计方法上下工夫。通过运用可开发性设计能显著提升存在可开发性问题项目的开发效率。

可开发性设计的本质是消除或减缓开发资源的瓶颈和开发环境的低效。



操作系统篇



操作系统是软件行业比较复杂的一个领域，我们只有具备深入理解操作系统的知识，才可能成为全面、专业的嵌入式软件工程师。为了向读者揭开操作系统的神秘面纱，本篇以循序渐进的方式讲解了 ClearRTOS“实时”操作系统是如何实现的。

嵌入式系统的启动是从引导加载器开始的，引导加载器在完成必要的系统初始化后，将加载嵌入式应用程序并将处理器交给它。第 19 章为读者概要地介绍了引导加载器的行为和作用。

任务是操作系统中的核心概念。在第 20 章将介绍什么是任务情景、任务调度器是什么、任务的切换原理等。此外，还介绍了检测任务栈溢出的方法和任务变量的用处与实现。

为了实现各任务间有效地协同工作，一定离不开使用信号量、互斥锁、事件和消息队列这些任务同步机制。在第 21 章就它们的实现进行了介绍。对于嵌入式软件开发，我们还得掌握互斥锁的优先级反转问题，以及理解如何通过优先级继承来解决该问题，这些内容也是第 21 章的范围。

第 22 章就堆管理和内存池管理进行了介绍，并就嵌入式软件中令人烦恼的内存泄漏问题提出了一种独特的解决方案。通过本章读者将完全明白堆分配与内存池分配的区别，也将了解如何通过设计去一步一步完善堆管理模块的实现。

嵌入式系统中的软件一定需要存取处理器的外设，在第 23 章介绍了如何通过构建设备访问模型的方式方便与外设进行交互。读完本章读者将对驱动程序有一定的理解。

在第 24 章讲解了软件定时器的实现原理，并解释了中断回调和任务回调定时器的区别。在本章还介绍了什么是实时性设计，并通过改进定时器模块实现的方式示例了如何提高设计的实时性。

第 25 章可以理解为第 20～24 章的总结。补充了 ClearRTOS 的设计原则，探讨了对它的一些改进意见，以及对它的展望。

第 19 章

引导加载器，系统启航者

在嵌入式系统中，通常将整个系统的软件分成两大部分，分别是引导加载器（boot loader）和应用程序[①]。注意：操作系统被包含在应用程序中。当然，引导加载器并不是必需的，因为完全可以将它与应用程序整合成一个“更大的”单体程序。但采用引导加载器有它的优点。

19.1 功能

正如名字所隐含的那样，引导加载器的第一个功能是引导系统运行。引导的结果是完成最小系统的初始化。最小系统是指为了实现引导加载器功能所需的最基本硬件和软件。初始化的内容有：

- 对处理器进行初始化。包括：处理器的工作时钟频率设置；各片选空间的时序和地址空间的配置；内存控制器的初始化；中断控制器的初始化；栈寄存器的初始化；等等。
- 对外设和相关的软件模块进行初始化。引导加载器都无一例外地需要提供控制台（console），以便实现人机交互，因此需要对串口硬件和命令行管理模块进行初始化；有的引导加载器，还支持通过以太网采用 TFTP 等协议进行应用程序加载，因而也需要对以太网硬件和 TFTP 协议栈进行初始化；等等。

引导加载器的第二个功能是，加载并执行应用程序。后面将就这一功能做进一步分析。

可被用于升级应用程序是引导加载器的另一个非常重要的功能。对于功能相对简单的引导加载器，可以采用串口控制台来实现对应用程序的升级。而复杂一点的引导加载器，就需要支持 TFTP 等协议实现通过以太网升级应用程序。应用程序的升级，可分为两大步骤：

- 将应用程序从主机传送到目标机内存中。传送的方式可以有多种，比如采用串口且基于 X-Modem 协议，或者采用以太网并基于 TFTP 协议，等等。各种传送方式通常都提供 CRC 等校验算法，以保证目标机上收到的文件是完整的。
- 将传输到目标机内存中的文件写入应用程序的外部存储介质中。

---

① 在基于 Linux 的嵌入式系统中，应用程序通常是独立于操作系统的，但这并不妨碍我们理解引导加载器。

除上面所涉及的功能外，不少引导加载器具备硬件诊断功能。通过诊断功能，可以检查硬件设备是否存在质量问题。

注意，引导加载器的功能越复杂，其在外部存储设备中所占用的空间就越大。对于存储资源受限的嵌入式系统，我们需要很好地权衡哪些功能应放入引导加载器中。

19.2　文件存储布局

由于引导加载器与应用程序是两个完全独立的程序文件，因此需要在外部存储空间中为它们分别分配不同的存储区域，图 19.1 示例说明了将两个文件分配在同一块闪存芯片上的情形。现实中，两者可以被存储于两块不同的闪存中，乃至一个存储在闪存中而另一个存储在硬盘内。

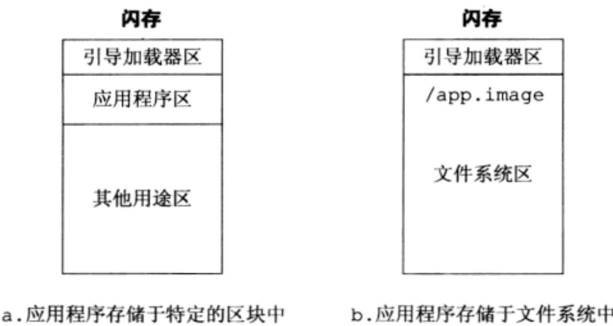


图 19.1

引导加载器与应用程序在文件格式上有所不同。引导加载器的格式通常被称为 BIN 文件，即二进制（binary）文件。引导加载器程序文件需要使用 objcopy 等工具，从一个 ELF 文件生成。与引导加载器不同的是，应用程序文件就是一个标准的 ELF 文件，是直接由编译器生成的可执行文件。

引导加载器一定不会通过文件系统进行存储，它的内容是直接存放在引导加载器区中的，即程序文件的第一个字节存放在引导加载器区的第一个字节处。应用程序却可以采用文件系统进行存储，当然也可以像引导加载器那样不使用文件系统。

当引导加载器需要加载应用程序时，如果应用程序也像引导加载器一样不通过文件系统存储，则直接从区块的第一个字节开始处读取应用程序文件即可。如果应用程序是存储于文件系统中的（即图 19.1 中的 b），那么引导加载器因为必须具备识别文件系统的能力而变得更复杂。在这种情形下，引导加载器必须通过文件操作的方式，打开并读取应用程序文件。

19.3 程序加载原理

在 1.3 节介绍了处理器是如何启动的。在采用引导加载器的嵌入式系统中，处理器上电复位时，所运行的第一条指令是由引导加载器提供的。也就是说，处理器最开始运行的是引导加载器程序。

对于应用程序文件，我们知道它是采用 ELF 格式进行存储的。在文件的开始处，存放的是 ELF 文件头，只有读取 ELF 文件头并分析后才知道.text 等段位于程序文件的哪一具体偏移处。但是，对于引导加载器就不能采用 ELF 文件格式进行存储，因为处理器上电运行引导加载器时，它并不认识什么是 ELF 文件格式。引导加载器程序是通过抽取编译器生成的 ELF 程序文件中的各段而组成的，这也是 BIN 文件与 ELF 文件最大的不同点。

对于一个将引导加载器放在闪存芯片上的系统来说（在本章的后面，都假设引导加载器是存储在闪存上的），处理器将从闪存中获取执行指令。其实，处理器并不知道自己所获取的指令来源于闪存，它只是从特定的地址处开始取指令，这是通过硬件设计实现的。在图 19.2 中，示例说明了处理器的程序计数器 PC（参见 1.1 节），在此时是指向位于闪存中的引导加载器程序的。另外，图中只象征性地示例说明了内存的地址，而没有示例说明闪存的地址，但读者应当意识到闪存也是占据处理器的一块地址空间的。

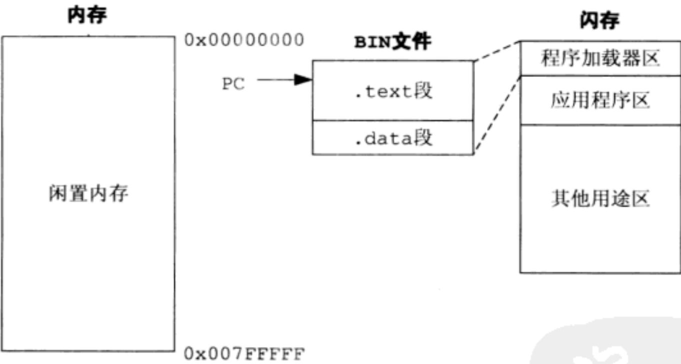


图 19.2

由于存取闪存的速度不如内存快，因此引导加载器会被设计成尽可能少地在闪存的地址空间中运行。当必要的初始化工作完成后，引导加载器就将自己拷贝到内存中，通过尽可能在内存中运行自己的方法提高执行效率。

自我拷贝是引导加载器很有趣的特点。为了做到这一点，它的.text 段可被分成两块，其中一块一直位于闪存中，而另一块却需要拷贝到内存中。引导加载器.text 段的可分块性需要在编写汇编程序时使用一定的技巧，具体的技巧需要结合特定的处理器进行解释，在此略过。

有的引导加载器为了节省外部存储空间，会对需要拷贝到内存中的那部分内容采用数据压缩的方式[②]。在这种情况下，引导加载器必须支持解压缩功能。

图 19.3 示例说明了引导加载器将自己在闪存中的内容拷贝到内存中的情形，图中还将本来是一体的.text 分为两部分，以方便表示其中一部分一直位于闪存中，另一部分则需要拷贝到内存中。.bss 段比较特殊，它并不需要从闪存中拷贝到内存中，而是只要在内存中开辟一块空间并对它进行置 0 初始化就行了。毫无疑问，引导加载器得知道自己的.bss 段在内存中的具体地址和大小，这通过使用链接器的链接脚本可以做到，在 6.2.2 节中已经介绍了这方面的内容。

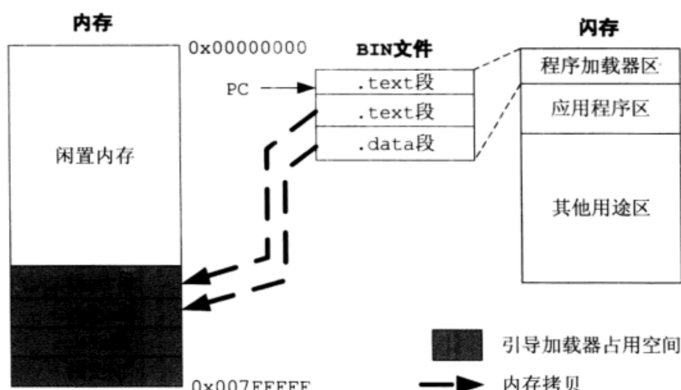


图 19.3

由于引导加载器的程序实现需要使用汇编语言和 C 语言，而 C 语言中的函数调用需要用到栈，因此在内存中需要开辟一块区域以用做程序调用的栈空间。这也正是为什么在图 19.3 中示例说明了栈空间的原因。开辟所需的栈空间，也是引导加载器的基本初始化工作之一。

在拷贝完成后，如何使处理器从内存开始无缝地（注意，必须是无缝地）运行后续指令，同样需要一定的编程技巧。对于这一内容的具体解释需要结合具体的处理器，在此略过。除了.text 段比较特殊，需要分为闪存部分和内存部分外，其他的.data 段、.bss 段都可以实现全部位于内存中。图 19.4 示例说明了程序计数器 PC 指向内存中的.text 段的情形。

引导加载器在进入内存运行后，将对相关硬件资源和软件模块做更进一步的初始化，内容包括中断向量表的初始化。中断向量表可以理解成一个数组，数组元素中存在一个函数指针域，当中断发生时，对应的函数指针所指向的函数（即 ISR，参见 1.6 节）将被处理器调用。每一类处理器都有自己特定的中断向量表格式，且可能定义了中断向量表在内存中的具体地址空间。图 19.5 示例说明了引导加载器初始化好中断向量表的情形，其中假设中断向量表位于内存的最开始处。

---

② 压缩操作是在编译的过程中完成的。

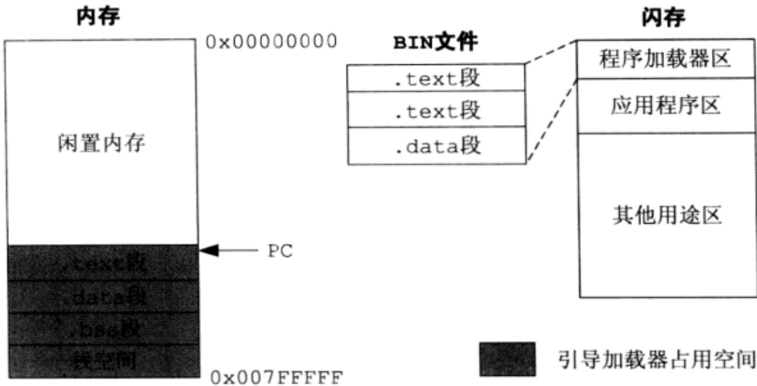


图 19.4

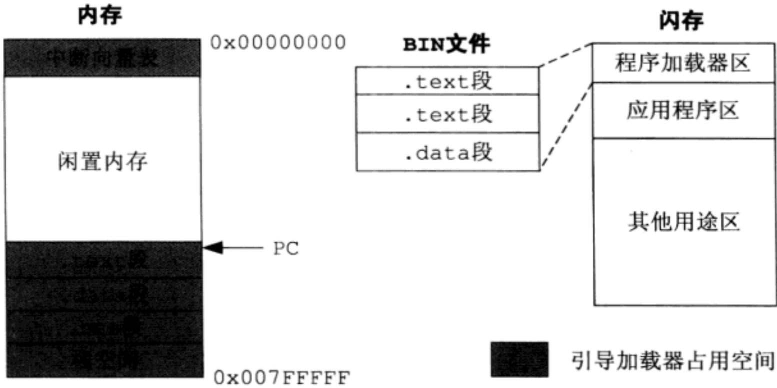


图 19.5

当所有的初始化工作完成后，引导加载器会监视并等待串口控制台几秒钟，看是否有操作人员通过敲击键盘加以干预，这向操作人员提供申请人机交互的机会。大多数引导加载器都设计有让操作人员通过串口终端（如 Windows 上的超级终端）中断引导加载器继续加载应用程序的功能。比如，操作人员可以通过按住 Esc 键的方式，使得引导加载器中断应用程序的加载，且等待用户在控制台输入命令以便进行应用程序升级等操作。

引导加载器等待期间，如果没有操作人员的干预其将自动加载应用程序。其做法是，从闪存的应用程序区或文件系统中的某一目录下读取 ELF 格式的应用程序文件，并根据 ELF 文件中的头信息，将程序的各段拷贝到指定的内存空间中。对于应用程序的.bss 段，程序加载器会直接对其内存空间清 0。

需要注意，引导加载器与应用程序所占用的内存空间不能有重叠，否则会造成应用程序加载后覆盖引导加载器程序的内容而使得引导加载器不能正常工作。规划引导加载器与应用程序

在内存中的布局，也是软件设计内容之一。图 19.6 展示了程序加载器对应用程序的加载，其中假设应用程序存在特定的区中，而不是文件系统中。注意：此时的程序计数器 PC 仍指向引导加载器的.text 段。

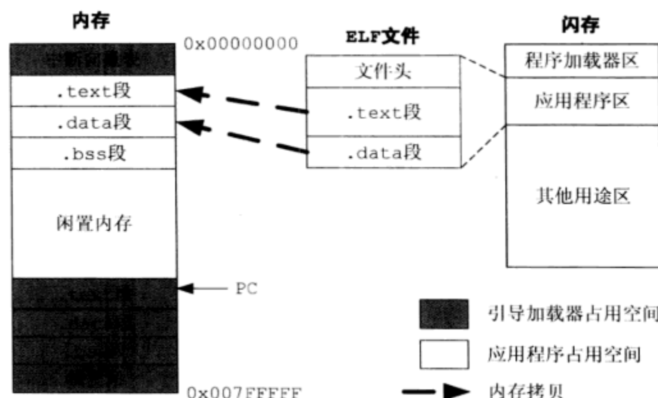


图 19.6

当引导加载器完成了对应用程序的加载后，它会根据应用程序文件中的 ELF 文件头信息获得应用程序的入口地址，以准备运行被加载的应用程序。很容易想到，应用程序的入口地址应当位于内存中应用程序的.text 段中。

获得应用程序的入口地址后，引导加载器会跳转到这一地址以运行应用程序。对于这一跳转动作，用专业一点的说法是 “引导加载器将控制权交给了应用程序”。如果应用程序中存在操作系统，则也可以说“引导加载器将控制权交给了操作系统”。一旦应用程序被运行，则所有的内存空间都属于应用程序，包括曾被引导加载器占用的空间。图 19.7 示例说明了应用程序刚运行时的情形。

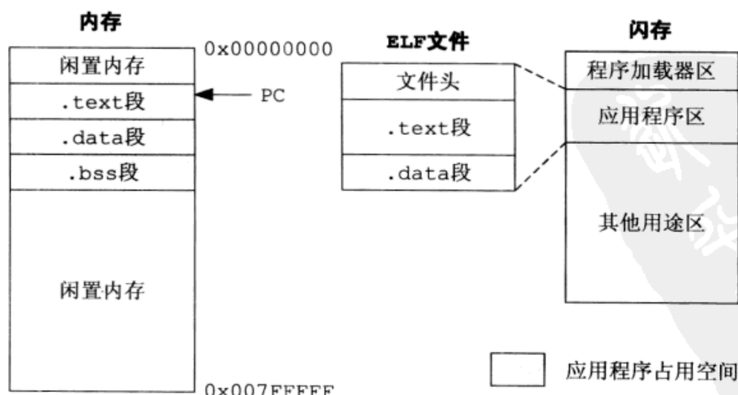


图 19.7

显然，应用程序中又得包含一部分初始化代码，以对应用程序的栈空间和处理器中断向量表等资源进行重新初始化。注意：中断向量表必须重新初始化，以指向位于应用程序中的.text段。由于大多的嵌入式系统应用程序与操作系统是合在一个程序中的，因此进行重新初始化的动作会由应用程序中的操作系统部分去完成。图 19.8 示例说明了应用程序完成重新初始化后的情形。

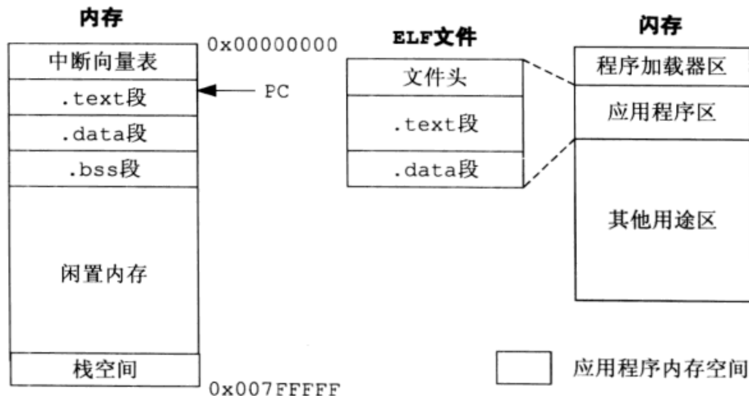


图 19.8

虽然这里没有就具体的处理器和引导加载器进行分析，但是不论是怎样的处理器和引导加载器，其基本思路和原理与这里所讲的几乎相同。

19.4　优点

引导加载器并不像应用程序那样需要经常变化，因为它的功能相对固定。通常，硬件设计一旦完成就大致决定了引导加载器的实现。

在使用引导加载器的系统中，升级应用程序时只更改应用程序所在的存储区，因此即使失败了也不会造成整个系统无法运行这种局面（至少引导加载器还能运行）。反之，在不采用引导加载器的系统中，就会出现一旦升级失败，整个系统就没有方便的恢复手段了，而是需要通过 JTAG 调试器或闪存烧写器等设备的辅助才能恢复系统。对于需要在公司之外进行升级的嵌入式系统来说，这是非常突出的一个优点。

19.5　小结

掌握引导加载器的工作原理，有助于系统性地理解嵌入式系统的软件是如何运行起来的。引导加载器的常见功能有：

- 实现处理器最小系统的初始化。

- 加载并运行应用程序。
- 升级应用程序。
- 对硬件设备进行诊断。



练习与思考

1. 引导加载器能存放于硬盘中以引导系统运行吗？

2. 为什么引导加载器不能存储于文件系统中以引导系统运行？

3. 引导加载器通常是采用汇编和 C 语言相结合进行编写的，那能不能全部用 C 语言来编写呢？

4. 当引导加载器在将控制权交给应用程序的过程中，处理器的中断应做怎样的处理？



第 20 章

任务，软件基本调度单元

从本章开始逐步介绍专为本书而设计的 ClearRTOS“实时”操作系统是如何实现的。为了展示 ClearRTOS 是如何演进的，以及每一次演进为了解决什么问题或提出什么概念，作者将演进的过程放入了光盘内 Project 目录下的 embedded 项目中。在第 25 章会在 embedded 项目的基础上形成 ClearRTOS 项目——ClearRTOS“实时”操作系统的最终完整版。

ClearRTOS 被设计成能在 Cygwin 环境和 Linux 操作系统上运行。这种方式一方面省去了读者为了学习而购买开发板，另一方面也便于读者修改源代码和调试。总之，便于读者实践。当然，这种方式使得 ClearRTOS 存在无法直接接管处理器的中断这一小小的局限，这使得 ClearRTOS 中的“滴答”需要通过使用 Linux 中的定时信号（signal）加以模拟，以及在 ClearRTOS 中无法实现在中断返回的过程中完成任务调度。正因为这两个局限点，使得我们讲 ClearRTOS 的“实时”时加上了引号。可以放心的是，这两个局限点丝毫不影响读者学习实时操作系统的设计和实现原理。

任务的概念与实现原理是学习操作系统首先要掌握的内容。在 1.5 节中指出，处理器只知道指令与数据，对任务的认识一无所知，任务是软件层面的抽象概念。但是，操作系统却需要实现“多个任务可以同时运行，且各任务可以从事完全不一样的工作并互不影响”。既然处理器并不知道任务这一概念，那多任务是如何实现的呢？我们先要从任务情景（task context）和任务调度（task schedule）说起。

在深入了解任务的细节之前，我们可以先看一看基于 ClearRTOS 实现的 taskv1 示例程序。读者需要先编译 embedded 项目，并运行其中的 taskv1.exe 可执行程序，图 20.1 示例说明了编译方法和示例程序的运行结果。embedded 项目编译环境的实现细节将从第 28 章到第 32 章进行介绍。

```
$ make

$ ./release/taskv1.exe

System is going to be up

Info: press Ctrl+C to terminate!
0

Error: stack overflow detected for task 1
```

```
12121212121212121212

Summary
-------
  Supported: 128
  Allocated: 4
  .BSS Used: 12436

Statistics
----------
    Task Scheduled: 32
  Stack Overflowed: 1
   Invalid Handle: 0
      Tick Delayed: 6

Task Details
------------
  Name: Idle
      Priority: 127
    Stack Base: 4255856
    Stack Size: 4096 bytes (45% used)
     Scheduled: 11
         State: RUNNING

  Name: 0
      Priority: 11
    Stack Base: 4243488
    Stack Size: 4096 bytes (overflowed)
     Scheduled: 1
         State: SUSPENDING

  Name: 1
      Priority: 16
    Stack Base: 4247584
    Stack Size: 4096 bytes (41% used)
     Scheduled: 10
         State: WAITING

  Name: 2
      Priority: 19
    Stack Base: 4251680
    Stack Size: 4096 bytes (41% used)
     Scheduled: 10
         State: WAITING

System is going to be down
```

图 20.1

taskv1.exe 示例说明了三个任务，它们的名字分别是“0”、“1”和“2”。各任务睡眠一定的时间，然后将自己的名字打印出来。另外，也演示了任务“0”出现栈溢出时能被调度器检测出来并将之挂起的行为。示例程序的输出结果中包含这些内容：

- 整体摘要。在摘要中指明了任务管理模块支持多少个任务、有多少个已分配任务，以及指明了任务模块所占用的.bss 空间大小。.bss 空间大小可以认为是任务模块的数据结构所占用的大致内存空间。

- 每一个任务的具体信息。包括任务的优先级、其栈空间在内存中的位置和大小、栈的利用率、被调度了多少次，以及任务状态

本章要求读者对第 10 章所介绍的栈帧概念有很好的掌握，因为任务切换需要在栈帧上做文章。

20.1　任务情景

一个任务一旦获得处理器而运行，是通过处理器的寄存器来获取指令并加工数据的。或者可以说，任务的行为是通过处理器的寄存器来体现的。任务情景正是指处理器中与任务运行相关的寄存器（的值）。

从处理器的角度来看，多任务的并行运行其实还是各任务一个接一个地串行运行的。当任务第一次获得运行机会时，任务管理模块会为之建立情景，当然情景是建立在处理器的寄存器中的。一旦任务因为某种原因要放弃处理器暂停运行时，任务管理模块所提供的函数会自动地将处理器上的任务情景保存到任务的私有内存中，这一动作被我们称为情景保存。当任务再一次获得处理器而运行时，任务管理模块又负责将保存在任务私有内存中的情景恢复到处理器的寄存器中，这一动作被我们称为情景恢复。每更换一次获得处理器而运行的任务包含情景保存和情景恢复两个动作，前一个动作是暂停正在运行的任务，而后一个动作则是让另一个任务获得运行机会。一次更替我们称之为情景切换。

20.1.1　情景内容

由于本书的 ClearRTOS 是运行在 Linux 操作系统之上的，所以任务情景所包含的寄存器更少。

在处理器不包含浮点运算单元的情形下，x86 处理器中的 EIP、EBP、ESP、EBX、EDI 和 ESI 六个寄存器需要被纳入到任务情景的范畴[①]。也就是说，在保存情景时，需要将这些寄存器的值写入到任务的私有内存中；而在恢复情景时，需要将保存在内存中的值恢复到这几个寄存器中。下面需要说一说这六个寄存器对于任务来说意味着什么。

EIP 是 x86 处理器的程序计数器，它保存了处理器下一条要运行的指令在地址空间中的位置。对于任务来说，它指示了任务“正在干什么”。显然，每一个任务的行为都可以不同，而任务切换时将这一寄存器的值保存起来就很容易理解了。从 C 语言的角度粗略地理解的话，可以将 EIP 理解为它记录了任务正在调用哪一个函数。

---

① 只包括这几个寄存器是因为在 Linux 操作系统和 Cygwin 环境中，ClearRTOS 不能直接接管处理器的中断，否则任务情景应当包含更多的处理器寄存器。另外，为了实现和理解上的简单性，ClearRTOS 中的任务不支持浮点处理，因此，任务情景中也不需要包含浮点寄存器。

光记住 EIP 还不行，因为它只指明了任务下一条要运行的指令是什么。函数间的调用离不开栈，由于每一个任务行为都不一样，所以各任务需要有自己独立的栈。在 x86 处理器中，栈基址寄存器 EBP 和栈指针寄存器 ESP 将被用于指示每一个任务的栈顶和当前栈帧，因此也需要将 EBP 和 ESP 纳入到情景的范畴中。

至于为什么 EBX、EDI 和 ESI 三个寄存器也需要保存，读者需要回忆在 10.3 节中所介绍的内容，从该节的图 10.4 中应当能找到答案。在 x86 处理器的 ABI 中，定义了 EBX、EDI 和 ESI 三个寄存器是可以用来存放函数的局部变量的，即最多可以有三个局部变量被存放在寄存器中。如果局部变量的数量多于三个，则多出来的部分将被保存在栈上，而 EBP 和 ESP 已经作为情景的一部分可以帮助任务在需要时恢复栈。

理解了在 x86 处理器上任务情景应当包含的寄存器后，就有了遍写任务情景处理函数的思路。

20.1.2　情景保存

因为需要操作处理器寄存器，情景保存函数需要通过编写汇编程序的方式来实现。图 20.2 是任务情景保存函数 context_save()的程序实现。

```
00032: #ifndef __register_t_defined
00033: typedef int register_t;
00034: #endif
00035:
00036: typedef struct {
00037:     register_t ebx_;
00038:     register_t esi_;
00039:     register_t edi_;
00040:     register_t ebp_;
00041:     register_t esp_;
00042:     register_t eip_;
00043: } general_purpose_registers_t;
00044:
00045: typedef struct {
00046:     general_purpose_registers_t gpr_;
00047: } task_context_t;
00048:
00059: int context_save (task_context_t *_p_context);
00060:
```

```
00029: // stack frame offset
00030: #define FRAME_OFFSET_PC        0
00031: #define FRAME_OFFSET_PARAM0    4
00032: #define FRAME_OFFSET_PARAM1    8
00033:
00034: // offset for registers of context
00035: #define CONTEXT_OFFSET_EBX     0
00036: #define CONTEXT_OFFSET_ESI     4
00037: #define CONTEXT_OFFSET_EDI     8
00038: #define CONTEXT_OFFSET_EBP     12
```

```
00039: #define CONTEXT_OFFSET_ESP     16
00040: #define CONTEXT_OFFSET_EIP     20
00041:
embedded/code/platform/arch/x86/[illegible]
00026: #include "offset.h"
00027:
00028:     .globl _context_save
00029:     .globl context_save
00030:     .align 4
00031:
00032: _context_save:
00033: context_save:
00034:     // get _p_context of context_save () for saving
00035:     movl FRAME_OFFSET_PARAM0(%esp), %edx
00036:
00037:     // save local variable registers into _p_context
00038:     movl %ebx, CONTEXT_OFFSET_EBX(%edx)
00039:     movl %esi, CONTEXT_OFFSET_ESI(%edx)
00040:     movl %edi, CONTEXT_OFFSET_EDI(%edx)
00041:
00042:     // save stack corresponding registers into _p_context
00043:     leal FRAME_OFFSET_PARAM0(%esp), %ecx
00044:     movl %ecx, CONTEXT_OFFSET_ESP(%edx)
00045:     movl %ebp, CONTEXT_OFFSET_EBP(%edx)
00046:
00047:     // save PC into _p_context
00048:     movl FRAME_OFFSET_PC(%esp), %ecx
00049:     movl %ecx, CONTEXT_OFFSET_EIP(%edx)
00050:
00051:     // return 0 to indicate a return from context_save ()
00052:     xorl %eax, %eax
00053:     ret
```

图 20.2

首先，在 context.h 中定义了用于保存任务情景的数据结构——task_context_t，其中包含了处理器的通用寄存器部分，由成员变量 gpr_表示。如果需要让 ClearRTOS 支持浮点运算，则 task_context_t 结构中还需要包含浮点运算寄存器部分。在 offset.h 中，定义了各寄存器变量在 general_purpose_registers_t 数据结构中的偏移量，这些偏移量在汇编代码中需要用到。

context_save()函数的原型可以从 context.h 的第 59 行找到。它的参数是一个类型为 task_context_t 的指针，指向的是用于保存情景的任务私有内存空间。

save.S 文件中示例说明了 context_save()函数的具体汇编程序实现。第 28～33 行以汇编的形式定义了 context_save()函数。这里定义了两个名称，一个是“context_save”，另一个则是“_context_save”，之所以为这个函数定义两个名称，是因为有些版本的编译器默认会在 C 程序中的函数名前加一个下划线，而有的则不会。为了让 ClearRTOS 适应不同版本的编译器，需要为这个函数定义两个名称。

从第 35 行开始，是 context_save()函数的具体实现。注意：context_save()函数并不创建自己的栈帧，后面会看到为什么需要这样。第 35 行获得_p_context 参数并放入 EDX 寄存器中，后面保存寄存器的操作都将以 EDX 寄存器为基址。在 offset.h 中，FRAME_OFFSET_PARAM0

被定义为 4。FRAME_OFFSET_PARAM0 是指第一个参数对 ESP 寄存器所指地址的相对偏移位置，FRAME_OFFSET_PARAM1 可以依此类推。至于第一个参数的偏移量为什么是 4，相信通过前面 10.4.1 节关于栈帧的学习可以理解。另外，也可以从后面讲解任务切换的图 20.6 中找到答案。

第 38～40 行分别将 EBX、ESI 和 EDI 寄存器存入到内存中。

第 43～45 行将栈相关的寄存器值保存到内存中。调用 context_save()函数（即汇编 call 指令所造成的压栈）而造成的压栈操作所带来的栈指针变化，并不应考虑在内，至于为什么后面还会涉及。x86 处理器中的 lea 指令[②]是用于获取有效地址的。

第 48 和 49 行则将程序计数器保存起来，FRAME_OFFSET_PC 在 offset.h 中被定义为 0，为什么是 0 同样可以从后面的图 20.6 中找到答案。第 52 行将 eax 中的值清为 0 以作为 context_save()函数的返回值。在 10.4.3 节中已指出整型返回值必须放入 EAX 寄存器中。最后第 53 行的 ret 指令[③]，将使得程序返回到 context_save()函数的调用点（的后一条指令）。

20.1.3　情景恢复

让一个任务获得处理器需要通过调用 context_restore()函数，其原型和实现可以从图 20.3 中找到。

```
[illegible]
00060: void context_restore (task_context_t *_p_context, int _value);
00061:
[illegible]
00026: #include "offset.h"
00027:
00028:     .globl _context_restore
00029:     .globl context_restore
00030:     .align 4
00031:
00032: _context_restore:
00033: context_restore:
00034:     // get _p_context of context_restore ()
00035:     movl FRAME_OFFSET_PARAM0(%esp), %ecx
00036:
00037:     // take 2nd param. of context_restore () as return value of context_save ()
00038:     movl FRAME_OFFSET_PARAM1(%esp), %eax
00039:
00040:     // restore registers from context
00041:     movl CONTEXT_OFFSET_EIP(%ecx), %edx
00042:     movl CONTEXT_OFFSET_EBX(%ecx), %ebx
00043:     movl CONTEXT_OFFSET_ESI(%ecx), %esi
00044:     movl CONTEXT_OFFSET_EDI(%ecx), %edi
00045:     movl CONTEXT_OFFSET_EBP(%ecx), %ebp
```

② lea 指令请参见参考资料[7]的第 648 页。

③ ret 指令请参见参考资料[8]的第 368 页。

```
00046:     movl CONTEXT_OFFSET_ESP(%ecx), %esp
00047:
00048:     // jump to context_save ()
00049:     jmp *%edx
```

图 20.3

context_restore()函数除了需要一个 task_context_t 类型的指针外，还得提供第二个参数作为 context_save()函数的返回值，在 20.1.4 节将解释这是如何做到的。

context_restore()函数的实现与 context_save()很相似，只是将_p_context 所指内存中的值放到寄存器中。不同点是，第 38 行将所传入的第二个参数放到 eax 寄存器中，且在函数的最后执行一个 jmp 指令[④]实现跳转（第 49 行）。FRAME_OFFSET_PARAM1 在 offset.h 中被定义为 8，从图 20.8 中可以理解它为什么是 8。

20.1.4　情景切换

context_switch()函数用于实现任务情景的切换，其实现如图 20.4 所示。函数的第一个参数所指内存用于保存被暂停任务的情景，第二个参数指向的内存中存放了将要运行任务的情景。

```
00099: void context_switch (task_context_t *_p_current,
           task_context_t *_p_next)
00100: {
00101:     if (0 == context_save (_p_current)) {
00102:         context_restore (_p_next, 1);
00103:     }
00104: }
```

图 20.4

要分析任务情景切换，需要获得 context_switch()函数的汇编代码，如图 20.5 所示[⑤]。请注意，必须使用调试版本的可执行程序来获得该函数所对应的汇编程序，因为非调试版本的程序会因为编译器的优化功能而变得“面目全非”，使得我们无法将 C 程序与汇编程序进行对照分析。为了方便讲解，假设存在 A 和 B 两个任务，且假设只存在从任务 A 切换到任务 B，以及从任务 B 切换到任务 A 两种情形。

```
$ make debug
$ objdump -S -d debug/taskv1.exe > dump.txt
$ vi dump.txt
...... 显示结果有删减 ......
void context_switch (task_context_t * p_current, task_context_t * p_next)
```

④ jmp 指令请参见参考资料[7]的第 619 页。

⑤ 读者自行获得的汇编代码最左边所显示的指令地址可能与这里列出的不一样，造成这种现象的原因可能是不同版本的编译器或操作系统。但无论如何，只是值不一样而已，并不影响我们分析问题。

```
{
  404805:       55                      push   ebp
  404806:       89 e5                   mov    esp, ebp
  404808:       83 ec 08                sub    $0x8, esp
   if (0 == context_save (_p_current)) {
  40480b:       8b 45 08                mov    0x8( ebp), eax
  40480e:       89 04 24                mov    eax,( esp)
  404811:       e8 32 02 00 00          call   404a48 <_context_save>
  404816:       85 c0                   test   eax, eax
  404818:       75 13                   jne    40482d <_context_switch+0x28>
      context_restore (_p_next, 1);
  40481a:       c7 44 24 04 01 00 00    movl   $0x1,0x4( esp)
  404821:       00
  404822:       8b 45 0c                mov    0xc( ebp), eax
  404825:       89 04 24                mov    eax,( esp)
  404828:       e8 3b 02 00 00          call   404a68 <_context_restore>
   }
}
  40482d:       c9                      leave
  40482e:       c3                      ret
```

图 20.5

现在假设是从任务 A 切换到任务 B，我们先分析任务情景切换中的保存部分。当需要进行任务调度（参见 20.2 节）时，context_switch()函数将会被调用，第一个参数指向的是用于保存任务 A 情景的内存，而第二个参数指向的内存则是任务 B 的。

图 20.4 中的第 101 行调用 context_save()函数，将处理器的寄存器值保留到任务 A 的私有内存中。根据前面 context_save()函数的实现，此时它将返回 0。从汇编的角度来看，图 20.5 中位于 404805～404808 处的指令用于创建函数的栈帧，404811 处的指令则调用 context_save()函数，call 指令[⑥]的调用会造成之后一条指令的地址（位于 404816 处）被压入栈中而形成图 20.6 所示的栈布局。

通过图 20.6 所示的栈布局，相信读者能更好地理解 20.1.2 节中 context_save()函数的实现。注意：在 context_save()函数中并不设立自己的栈帧，因此不会更改图中 ESP 寄存器的值，而此图也很好地说明了为什么在 context_save()函数中_p_current 参数的位置是在 ESP 所指位置加 4 的地方。还得注意：在 context_save()函数中，此时 ESP 寄存器所指栈单元的值会保存到任务 A 私有内存中的 eip_变量内。

另外，图 20.6 中也示例说明了被保存到任务私有内存中 esp_变量的值，它的值并不是此时 ESP 寄存器的值，而是除去了返回地址 404816 在栈中所占用的空间。为什么这样后面还会做进一步解释。

当 context_save()函数在最后调用 ret 指令后，保存在栈顶的返回地址将被弹出到 EIP 寄存器中，结果就是程序将从 404816 位置处继续运行，此时的栈布局如图 20.7 所示。由于

⑥ call 指令请参见参考资料[7]的第 168 页。

context_save()函数返回 0，这将导致 484818 位置处的 jne 指令[7]并不使程序跳转，而是继续从 40481a 位置处运行，即接着调用 context_restore()函数。

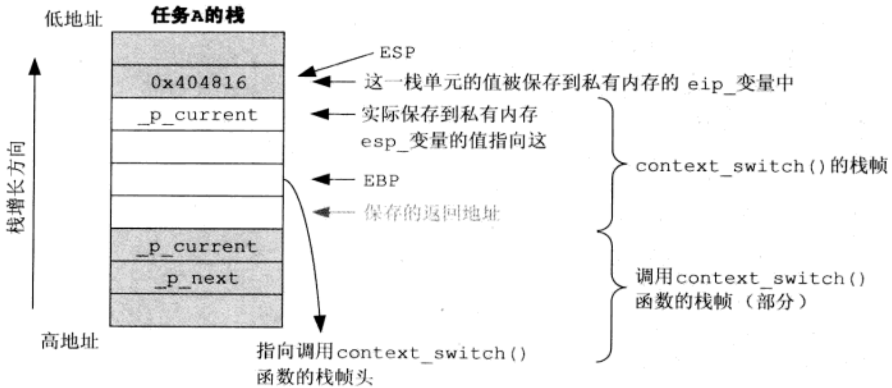


图 20.6

在继续讲解 context_restore()函数被调用之前，需要假设是从任务 B 切换到任务 A 的情形。之所以又做这一假设，是为了让读者在考察情景切换时，始终站在一个任务的角度，即这里的任务 A。基于这一假设，在 context_switch()调用 context_restore()函数时，_p_next 是指向任务 A 的私有内存的。注意：_p_next 所指内存中的内容是以前从任务 A 切换到任务 B 时通过 context_save()函数保存的，其中 eip_变量的值是 404816。

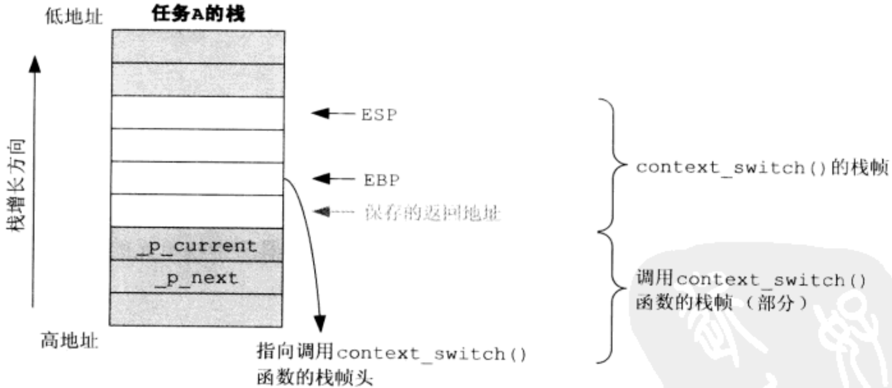


图 20.7

在 context_switch()函数调用 context_restore()函数时，将形成图 20.8 所示的内存布局。注意：此时栈上保存的返回地址是 40482d，且此时的栈空间是任务 B 的，因为是从任务 B 切换到任务 A。

⑦ jne 指令请参见参考资料[7]的第 611 页。

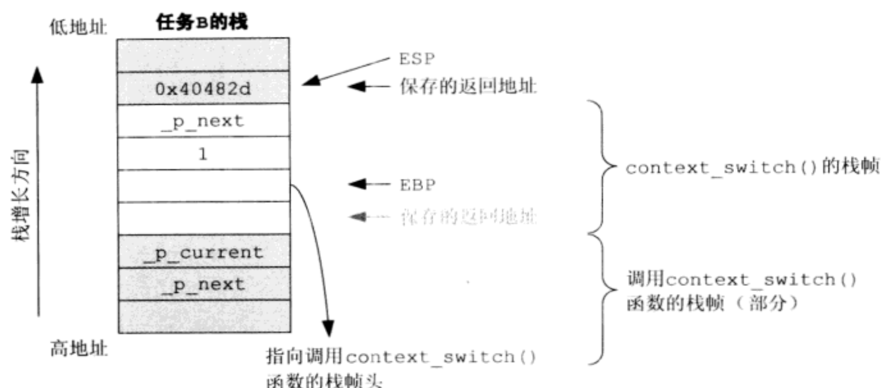


图 20.8

在 context_restore()函数中，任务 A 的情景将从_p_next 所指向的内存中恢复到处理器的寄存器内，且所传给它的第二个为 1 的参数也将被放入到 EAX 寄存器内。从图 20.3 中可以看出，在 context_restore()函数的最后，eip_变量所保存的值将会被放入 EDX 寄存器中，并在最后跳转到对应的位置处，即跳转到内存地址 404816 处。context_restore()函数调用完后，将形成与图 20.7 一样的栈空间布局。请注意，此时的栈空间已经变成了任务 A 的，ESP 所指位置是由图 20.6 中的"实际保存的 ESP 值"所决定的。

在 x86 处理器中，并不提供直接改变程序计数器值的指令，它必须通过跳转指令 jmp、调用指令 call 或返回指令 ret 间接地做到。注意：在 context_save()函数的最后，使用的是 ret 指令进行返回，ret 指令会造成处理器进行一次退栈操作，并从 ESP 寄存器所指位置弹出将要运行的指令。在 context_restore()函数的最后，并没有使用 ret 指令，而是采用 jmp 指令。由于 jmp 指令并不对栈产生任何影响，这也正是为什么在 context_save()函数中保存栈顶指针时，不是直接使用当时 ESP 寄存器中的值的原因。

很明显，在 context_restore()函数调用完后，任务 A 的情景就完全恢复了。跳转以后，程序将回到 404816 位置处继续运行，且再一次对 EAX 寄存器中的值是否为 0 进行判断。这一次，由于在 context_restore()中，将 EAX 寄存器中的值变为了 1，所以测试结果不为 0，也就是说，程序将跳转到 40482d 处继续运行。40482d 处的 leave 指令[8]用于"抹去"context_switch()函数的栈帧为返回做准备。

context_save()和 context_restore()通过巧妙的设计，使得对于同一个任务在被暂停和重新获得处理器之后，如同没有图 20.4 中的第 101～103 行的代码一样。

掌握了任务情景这一概念，以及明白任务情景切换的机理后，接着看一看任务调度。

⑧ leave 指令请参见参考资料[7]的第 651 页。

20.2 任务调度

安排各任务有序运行的行为被称为任务调度，而实现这一功能的程序被称为任务调度器。任务调度的基本思路是，根据一定的选择标准，找到下一个将要运行的任务，接着在正在运行的任务与将要运行的任务之间做一次情景切换。下一个运行任务的选择标准就是调度算法。

20.2.1 调度算法

在所有的调度算法中，主要有基于任务优先级和基于时间片两种算法，或者是这两种方式的组合。

20.2.1.1 优先级和时间片

对于基于优先级的算法，每一个任务在创建时都需要为之指定一个优先级。任务调度器在每一次需要调度时，根据任务的优先级选择出最高优先级的任务，并将处理器分配给该任务使其运行起来。基于优先级算法的最大好处是，能最大程度地满足时间响应速度要求高的事务处理，而这也是实时嵌入式软件的一个很重要的特点。正因如此，所有的实时操作系统都存在基于优先级调度的功能，这一点 ClearRTOS 也不例外。在基于任务优先级的系统中，对实时性要求高的任务在创建时可以为之分配高优先级。

基于时间片的调度方式是规定一个任务在一次获得处理器时最多能运行多长时间，当一个任务的规定时间用完了以后，调度器会强行剥夺任务的运行权。这种调度算法的好处是，不容易存在因为某一个任务太“贪婪”，而造成其他的任务无法获取处理器而“饿死”。

基于优先级和基于时间片这两种调度算法都有各自的优点，因此在很多操作系统的实现中会考虑同时采用它们，且通过运用这两种算法的不同组合衍生出不同的调度模式。在 ClearRTOS 中，只采用基于优先级的调度算法，且每一个优先级只能属于一个任务。

20.2.1.2 算法实现

位图在很多地方被用做数据管理的算法，比如在 UNIX/Linux 操作系统的文件系统就采用位图来管理磁盘扇区。在 ClearRTOS 中，同样可以采用位图来实现快速地查找优先级最高的任务。从模块化设计的角度来看，位图管理可以当做独立的一个模块，相关数据结构的定义如图 20.9 所示。

```
00042: #define CONFIG_MAX_BITMAP_ROW        8
00043: #define CONFIG_MAX_BIT_PER_ROW       8
00044:
00045: #define BITS_SUPPORTED (CONFIG_MAX_BITMAP_ROW*CONFIG_MAX_BIT_PER_ROW)
00046: #define LAST_BIT        (BITS_SUPPORTED - 1)
00047:
00048: #define INVALID_BIT ((bit_t)-1)
```

```
00049:
00050: typedef u8_t task_bitmap_row_t;
00051: typedef u8_t bit_t;
00052:
00053: typedef struct {
00054:     task_bitmap_row_t buffer_ [CONFIG_MAX_BITMAP_ROW];
00055:     task_bitmap_row_t row_bitmap_;
00056: } task_bitmap_t, *task_bitmap_handle_t;
```

图 20.9

对于位图模块的设计实现，首先要理解的是 task_bitmap_t 数据结构。通过对 bitmap.h 中第 42 和 43 行的两个宏，以及第 50 行定义的 task_bitmap_row_t 类型，可以实现对任务位图模块进行灵活配置，这有助于节约内存。图 20.9 中的配置支持 64 个任务优先级，图 20.10 示例说明了该 task_bitmap_t 结构的内存布局。

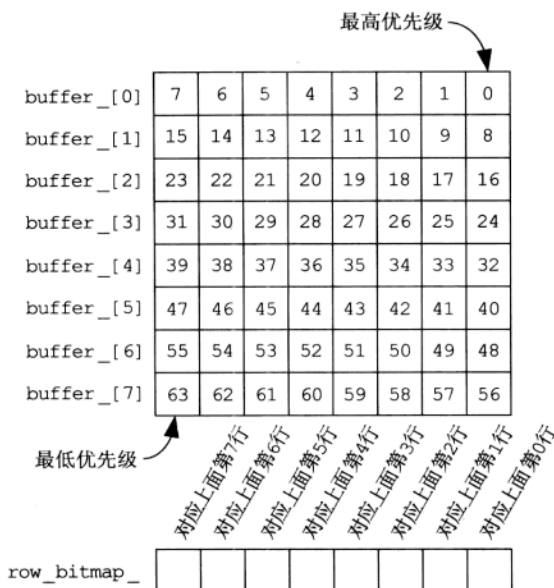


图 20.10

task_bitmap_t 中的 buffer_数组中的每一位对应一个优先级，且优先级是从 0 开始的。在 ClearRTOS 中，0 表示最高优先级。

row_bitmap_成员变量的定义是为了实现位图的高效查找，其中的每一个比特代表对应的 buffer_数组元素中是否存在值为 1 的比特位，值为 1 表示存在。图 20.10 中也列出了 row_bitmap_成员变量中各比特所对应的数组元素。

如果图 20.10 所示的任务位图用于表示就绪状态的任务，那如何在每一次任务调度时通过这个位图获得其中的最高优先级呢？这可以通过查表法来实现。

对于一个字节，其存在 256 个可能值。从任务优先级的角度来看，每一个值都可以找到其中优先级最高的比特。比如值 49（二进制格式是 110001）时最高优先级为 0，48（二进制格式是 110000）时为 4。我们可以建立如图 20.11 所示的查找表，来查找一个字节中的最高优先级。表中，对应于 0 值所返回的优先级是 0xFF，即表示无效的优先级，在代码中用 INVALID_BIT 宏表示。

embedded/code/platform/task/v1/src/bitmap.c

```
00063: static unsigned char g_bitmap_table [] = {
00064:     0xFF,    0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
00065:     4,       0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
00066:     5,       0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
00067:     4,       0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
00068:     6,       0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
00069:     4,       0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
00070:     5,       0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
00071:     4,       0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
00072:     7,       0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
00073:     4,       0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
00074:     5,       0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
00075:     4,       0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
00076:     6,       0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
00077:     4,       0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
00078:     5,       0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
00079:     4,       0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
00080: };
```

图 20.11

为了避免对 buffer_数组的遍历，可以通过 row_bitmap_变量，并同样通过查表的方式，找到最高优先级在 buffer_数组中的索引号。图 20.12 示例说明了用于从任务位图中获取最高优先级函数 task_bitmap_lowest_bit_get()的实现。图中的 bitmap_to_bit()函数用于从给定的_bitmap 中，找到其中不为 0 的最低比特（或者说是优先级最高的比特）。

embedded/code/platform/[illegible]

```
00083: static inline bit_t bitmap_to_bit (u32_t _bitmap)
00084: {
00085:     int bitmap_byte;
00086:
00087:     if (0 == _bitmap) {
00088:         return INVALID_BIT;
00089:     }
00090:
00091:     bitmap_byte = _bitmap & 0xFF;
00092:     if (0 != bitmap_byte) {
00093:         return g_bitmap_table [bitmap_byte];
00094:     }
00095:
00096:     bitmap_byte = (_bitmap & 0xFF00) >> 8;
00097:     if (0 != bitmap_byte) {
00098:         return g_bitmap_table [bitmap_byte] + 8;
00099:     }
00100:
00101:     bitmap_byte = (_bitmap & 0xFF0000) >> 16;
00102:     if (0 != bitmap_byte) {
```

```
00103:         return g_bitmap_table [bitmap_byte] + 16;
00104:     }
00105:
00106:     bitmap_byte = (_bitmap & 0xFF000000) >> 24;
00107:     return g_bitmap_table [bitmap_byte] + 24;
00108: }
00109:
00111: bit_t task_bitmap_lowest_bit_get (const task_bitmap_handle_t _handle)
00112: {
00113:     bit_t row, bit;
00114:
00115:     row = bitmap_to_bit (_handle->row_bitmap_);
00116:     if (INVALID_BIT == row) {
00117:         return row;
00118:     }
00119:
00120:     bit = bitmap_to_bit (_handle->buffer_[row]);
00121:     bit += row << g_bits_per_row;
00122:     return bit;
00123: }
00124:
00126: bool task_bitmap_is_empty (const task_bitmap_handle_t _handle)
00127: {
00128:     return (bool)(0 == _handle->row_bitmap_);
00129: }
```

图 20.12

task_bitmap_lowest_bit_get()函数的实现很直接，首先从 row_bitmap_中找到最高优先级在 buffer_数组中的索引（第 115 行），然后从 buffer_对应的数组元素中找到最高优先级的比特位（第 120 行），并在最后进行适当的移位操作（第 121 行），以得到最终的最高优先级。g_bits_per_row 用于表示 buffer_数组中一个元素（或一行）的比特位数，这一变量在任务位图初始化时被设置。

另外，图中还示例说明了 task_bitmap_is_empty()函数的实现，当位图中全部比特位为 0 时该函数返回 true。

图 20.13 示例说明了任务位图模块其他函数的实现。task_bitmap_init()函数用于对位图进行初始化，初始化的过程包含对模块的全局变量 g_bits_per_row 进行初始化。对于 8 比特的 task_bitmap_row_t，g_bits_per_row 应被初始化为 3，表示对 1 左移 3 个比特就获得 8 这个值。对 g_bits_per_row 变量的初始化，是通过调用 convert_to_shift_bits()函数做到的，这个函数的实现可以从 alignment.c 文件中找到。

```
00034: static int g_bits_per_row;
00035:
00036: void task_bitmap_init (task_bitmap_handle_t _handle)
00037: {
00038:     _handle->row_bitmap_ = 0;
00039:     memset (_handle->buffer_, 0, sizeof (_handle->buffer_));
00040:     (void) convert_to_shift_bits (CONFIG_MAX_BIT_PER_ROW, &g_bits_per_row);
00041: }
00042:
```

```
00043: void task_bitmap_bit_set (task_bitmap_handle_t _handle, bit_t _bit)
00044: {
00045:     bit_t row = _bit >> g_bits_per_row;
00046:     bit_t bit = (bit_t)1 << (_bit & (CONFIG_MAX_BIT_PER_ROW - 1));
00047:
00048:     _handle->buffer_ [row] |= bit;
00049:     _handle->row_bitmap_ |= ((bit_t)1 << row);
00050: }
00051:
00052: void task_bitmap_bit_clear (task_bitmap_handle_t _handle, bit_t _bit)
00053: {
00054:     bit_t row = _bit >> g_bits_per_row;
00055:     bit_t bit = (bit_t)1 << (_bit & (CONFIG_MAX_BIT_PER_ROW - 1));
00056:
00057:     _handle->buffer_ [row] &= ~bit;
00058:     if (0 == _handle->buffer_ [row]) {
00059:         _handle->row_bitmap_ &= ~((bit_t)1 << row);
00060:     }
00061: }
```

图 20.13

task_bitmap_bit_set()和 task_bitmap_bit_clear()两个函数分别用于对位图中的比特位进行置 1 和清 0，两个函数的实现同样很直接，这里无须解释更多。

20.2.2 调度器

需要通过调用 task_schedule()函数来触发调度器进行任务调度，它的实现代码示于图 20.14 中。

```
00049: typedef void (*preschedule_callback_t) (task_handle_t _from, task_handle_t _to);
00050:
00052: typedef struct {
00053:     statistic_t scheduled_;
00054:     statistic_t overflowed_;
00055:     statistic_t invalid_handle_;
00056: } task_statistic_t;
00057:
00059: task_bitmap_t g_ready_bitmap;
00063: static int g_scheduler_locked_count;
00067: static task_statistic_t g_statistics;
00069: static task_handle_t g_task_running;
00074: static bool g_multitasking_started;
00075:
00184: void task_schedule (preschedule_callback_t _callback)
00185: {
00186:     interrupt_level_t level;
00187:     task_handle_t running, successor;
00188:     bool overflowed = false;
00189:
00190:     level = global_interrupt_disable ();
00191:     if (!g_multitasking_started) {
00192:         global_interrupt_enable (level);
00193:         return;
```

```
00194:     }
00195:     if (is_in_interrupt () || (g_scheduler_locked_count != 0)) {
00196:         global_interrupt_enable (level);
00197:         return;
00198:     }
00199:     successor = g_priority_map [task_bitmap_lowest_bit_get (&g_ready_bitmap)];
00200:     if (successor == g_task_running) {
00201:         global_interrupt_enable (level);
00202:         return;
00203:     }
00204:     running = g_task_running;
00205:     g_task_running = successor;
00206: ......
00214:     if (TASK_STATE_RUNNING == running->state_) {
00215:         (void) task_state_change (running, TASK_STATE_READY);
00216:     }
00217:     (void) task_state_change (successor, TASK_STATE_RUNNING);
00218:     g_statistics.scheduled_ ++;
00219:     successor->stats_scheduled_ ++;
00220:     if (null != _callback) {
00221:         _callback (running, successor);
00222:     }
00223:     context_switch (&running->context_, &successor->context_);
00224:     global_interrupt_enable (level);
00225: }
```

图 20.14

先看一看 task.c 文件。第 52～56 行定义了任务模块所需的统计信息数据结构。第 67 行则定义了这一类型的一个全局变量以记录任务模块的统计信息。在第 59 行定义了 g_ready_bitmap 变量，用于记录整个系统中所有处于“就绪”状态任务的优先级。第 63 行的 g_scheduler_locked_count 变量用于记录调度器被嵌套锁定的次数，值为 0 表示调度器没有被锁定，后面马上还会讨论这个变量的用处。第 69 行定义了 g_task_running 变量用于记录正在运行的任务，task_handle_t 数据结构将在后面介绍。第 74 行的 g_multitasking_started 变量用于标识系统是否已经处于多任务状态。

第 184～225 行是 task_schedule()函数的实现。这个函数只有一个参数，参数的类型为 preschedule_callback_t，被定义于 task.h 中的第 49 行，它是一个函数指针类型。参数_from 代表将要被暂停的任务，而_to 则代表将要运行的任务。

第 187 行定义了两个局部变量，其中 running 变量用于记录调度之前正在运行的任务，而 successor 变量中记录的是调度完成后将要运行的任务。第 190 行则是在进行任务调度之前，关闭处理处理器上的所有可屏蔽中断以防出现竞争问题，在 20.5 节就中断与任务管理做了更加系统的讨论。为了简化起见，后面在介绍程序实现时将忽略所有对控制中断的代码。

第 191～194 行查看多任务环境是否已启动，如果多任务环境没有被启动，则调用 task_schedule()函数是无效的，如何进入和退出多任务环境是后面 20.9 节的话题。

task_schedule()函数并不允许在中断状态下被调用，在中断状态下触发任务调度需要通过 task_schedule_in_interrupt()函数，20.6 节将进一步探讨中断下的任务调度问题。另外，当调度

器被锁定时，task_schedule()函数也不应当做真正的任务调度工作。第 195～198 行代码的作用正是针对这两种情形的。

第 199 行从 g_priority_map 映射数组中获得系统中处于“就绪”状态的最高优先级任务，这里用到了 task_bitmap_lowest_bit_get()函数。第 200 行将获得的任务与 running 变量进行比对，看是否真的需要一次任务调度。如果两个变量的值相同，则说明并不需要进行任务调度，此时 task_schedule()函数可以直接返回（第 202 行）。

程序运行到第 204 行，说明真的有必要做一次任务切换。第 204 行记录下将要被暂停的任务，第 205 行则把将要运行任务的赋值给 g_task_running 变量。第 214 行检查将要被暂停的任务是否处于“运行”状态[⑨]。如果被暂停的任务处于运行状态，则需要调用 task_state_change()函数将它变为“就绪”状态。

第 217 行对将要运行的任务调用 task_state_change()函数，将其状态变为“运行”。第 218 行则对系统所进行过的任务调度次数进行统计。第 219 行统计单个任务被调度了多少次。第 220 行检查_callback 参数是否是有效的，如果有效则调用它所指向的回调函数。在第 223 行调用 context_switch()函数进行任务情景切换，以便最终完成任务调度。最后，第 224 行恢复进入 task_schedule()函数时的中断状态。

在任务管理模块中，为了防止出现竞争问题，在关键的代码片段中需要通过关闭中断的方式来防止。但是，中断不能长时间被关闭，因为这会影响处理器对外部中断的响应。当只希望防止任务间的竞争问题而非任务与中断间的竞争问题时，可以考虑通过锁定调度器的方式来解决。图 20.15 是对调度器进行锁定与解锁函数的程序实现。

```
embedded/code/platform/[illegible]
00612: void scheduler_lock ()
00613: {
00614:     interrupt_level_t level;
00615:
00616:     if (is_in_interrupt ()) {
00617:         return;
00618:     }
00619:
00620:     level = global_interrupt_disable ();
00621:     if (!g_multitasking_started) {
00622:         global_interrupt_enable (level);
00623:         return;
00624:     }
00625:     g_scheduler_locked_count ++;
00626:     global_interrupt_enable (level);
00627: }
00628:
00629: void scheduler_unlock ()
00630: {
00631:     interrupt_level_t level;
```

⑨ 任务调度有可能是因为某一任务删除自己而触发的，在这种情形下并不需要将被暂停的(已删除)任务的状态更改为“就绪”。

```
00632:
00633:     if (is_in_interrupt ()) {
00634:         return;
00635:     }
00636:     level = global_interrupt_disable ();
00637:     if (!g_multitasking_started) {
00638:         global_interrupt_enable (level);
00639:         return;
00640:     }
00641:     g_scheduler_locked_count --;
00642:     global_interrupt_enable (level);
00643:
00644:     task_schedule (null);
00645: }
```

图 20.15

调度器的锁定与解锁动作不允许发生在中断状态，因此 scheduler_lock()和 scheduler_unlock()函数在它的开始处都会判断函数是否是在中断状态被调用，如果是则直接返回。第 621 和 637 行分别检查系统是否处于多任务状态，如果不是则对于调度器的锁定和解锁就毫无意义，此时函数可以立即返回。

scheduler_lock()与 scheduler_unlock()函数是对变量 g_scheduler_locked_count 进行操作，锁定时进行加一操作，反之进行减一操作。scheduler_unlock()函数在最后需要调用 task_schedule()函数触发一次任务调度。之所以需要触发一次任务调度，是因为在调度器被锁定期间，有可能有优先级更高的任务因为中断的发生而处于就绪状态。从图 20.14 的第 195 行可以发现，当 g_scheduler_locked_count 变量的值不为 0 时，即调度器仍处于被锁定状态时，scheduler_unlock()函数调用 task_schedule()函数所触发的任务调度并不会真正发生。

采用变量计数的方式来管理调度器的锁定，可以有效地解决调度器被重复锁定的问题。这种方式还被用于管理中断的嵌套（参见 23.2.3 节）。

需要注意，锁定调度器的动作最好发生在操作系统的内核代码中，这一行为应避免出现在应用程序代码中。

20.3 任务的生命周期

任务的生命周期始于被创建，而终于被删除或任务自行退出。要了解任务的生命周期，必须了解任务在生命周期中的状态变迁。在本节，我们只关注任务管理模块的内部函数对任务状态变迁的影响。至于信号量、互斥锁、事件和消息队列等对任务状态的影响，将在后续的相关章节中再讨论。

ClearRTOS 中任务状态的变迁列于图 20.16 中。后面，在讲解任务相关函数的实现时，可以参照这一状态图辅助理解。

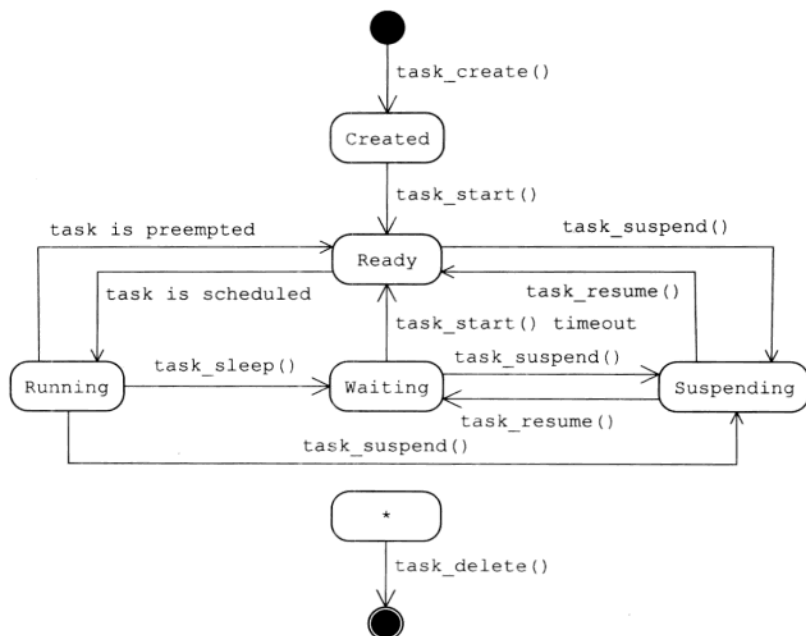


图 20.16

创建任务是通过调用 task_create()函数实现的，任务标识（类型为 task_handle_t）也会因为这个函数的调用而获得。任务一旦创建就处于“已创建（Created）”状态。

对已创建的任务调用 task_start()函数将使其进入“就绪（Ready）”状态。一旦任务处于“就绪”状态，调度器才“考虑”对其进行调度。处于“就绪”状态的任务，其中优先级最高的任务将被调度器调度运行而进入“运行”状态；处于“就绪”状态的任务，也可能因为其他任务调用 task_suspend()函数而使其被挂起从而进入“挂起（Suspending）”状态。

一个处于“运行”状态的任务，也可能因为系统中有更高优先级的任务就绪而被抢占从而进入“就绪”状态。另一方面，一个处于“运行”状态的任务，也可以因为调用 task_sleep()函数而让自己进入“等待（Waiting）”状态，或者因为 task_suspend()函数的被调用而进入“挂起”状态。

处于“等待”状态的任务，当所希望等待的时间到期后，任务将从“等待”状态转换到“就绪”状态。一个任务在等待期间也可以被其他任务调用 task_suspend()函数而进入“挂起”状态。

当任务处于“挂起”状态时，只有被其他的任务调用 task_resume()函数，才能使其退出“挂起”状态。至于任务退出挂起状态后进入哪一个状态，完全由任务在被挂起时的状态所决定，它可能进入“等待”或“就绪”状态。

被创建的任务无论处于何种状态，通过调用 task_delete()函数可以终止其生命。在状态图

中，“已删除（Deleted）”并不是一个状态，但从软件实现的角度来看，需要这个状态用于表示一个任务已被删除。

20.4　任务控制

对任务的控制，是通过相应的函数来做到的。在介绍 ClearRTOS 中的任务控制函数之前，需要先了解用于管理任务的数据结构，如图 20.17 所示。

```
00035: typedef u32_t stack_unit_t;
00036:
00037: #ifndef __task_priority_defined__
00038: typedef u32_t task_priority_t;
00039: #define __task_priority_defined__
00040: #endif
00041:
00042: #ifndef __task_handle_defined__
00043: struct type_task;
00044: typedef struct type_task task_t, *task_handle_t;
00045: #define __task_handle_defined__
00046: #endif
00047:
00048: typedef void (*task_entry_t) (const char _name [], void *_p_arg);
00050:
00051: #define STACK_DECLARE(_name, _size) static stack_unit_t _name [_size]
00052:
00053: typedef enum {
00054:     TASK_STATE_CREATED = 0x001,
00055:     // task is ready to be scheduled
00056:     TASK_STATE_READY = 0x002,
00057:     TASK_STATE_RUNNING = 0x004,
00058:     TASK_STATE_SUSPENDING = 0x008,
00059:     // task is waiting for timeout, semaphore or mutex
00060:     TASK_STATE_WAITING = 0x010,
00061:     TASK_STATE_DELETED = 0x020
00062: } task_state_t;
00063:
00064: // task control block
00065: struct type_task {
00066:     dll_node_t node_;
00067:     magic_number_t magic_number_;
00068:
00069:     task_context_t context_;
00070:     task_state_t state_;
00071:     timer_handle_t timer_;
00072:     msecond_t timeout_;
00073:     error_t ecode_;
00074:
00075:     // statistics
00076:     statistic_t stats_scheduled_;
00077:
00078:     // user provided information
00079:     task_priority_t priority_;
00080:     address_t stack_base_;
00081:     usize_t stack_size_;
```

```
00082:     task_entry_t entry_;
00083:     void *argument_;
00084:     char name_ [NAME_MAX_LENGTH + 1];
00085: };
```

```
00042: #define MAGIC_NUMBER_TASK         0x5441534BL
00043:
00049: #define is_invalid_handle(_handle) (((_handle) == null) || \
00050:     ((_handle)->magic_number_ != MAGIC_NUMBER_TASK))
00051:
```

图 20.17

task.h 的第 35 行定义的 stack_unit_t 是表示一次栈操作所对应的数据类型[⑩]。在 32 位处理器上，每次栈操作都会造成栈指针加或减 4 个字节，为此 stack_unit_t 被定义为大小为 4 个字节的类型。第 42～46 行通过使用 typedef 将第 65 行定义的 type_task 结构重新定义为 task_t，以及 task_handle_t，后者是 task_t 的指针类型。显然，task_t 和 task_handle_t 都可用于代表一个任务。第 48 行定义了每一个任务体函数的函数原型，任务体函数与任务的关系就如同 main() 入口函数与 C 程序的关系。第 51 行定义了 STACK_DECLARE 宏用于为任务定义栈空间。在 ClearRTOS 中，任务的栈内存通过使用 STACK_DECLARE 宏定义全局静态数组的形式创建。第 53～62 行定义了任务的所有可能状态。

第 65～85 行定义了任务的管理数据结构，这一数据结构在有的操作系统中又被称为任务控制块（Task Control Block，TCB）。上面提到，这一结构又被重定义为 task_t 和它的指针类型 task_handle_t。这一结构中各变量的作用如下：

- node_：每一个被分配出来的任务，通过链表的形式组织在一起，这个变量正是充当链表结点的功能。
- magic_number_：任务管理数据结构的有效性非常重要，通过为任务数据结构分配一个固定的识别值，并在结构初始化时保存在 magic_number_变量中，以辅助有效性检查。
- context_：用于保存任务的情景。
- state_：用于记录任务的状态。
- timer_：每个任务都对应有一个软件定时器（参见第 24 章），供任务进入“等待”状态时使用。
- timeout_：这个变量被当做是过渡参数，在任务函数之间传递，其中记录的是任务所需等待的时间，用于指定任务的等待超时值。
- ecode_：用于在必要的情况下记录任务从等待状态变为就绪状态时的返回错误码。
- stats_scheduled_：记录任务被调度了多少次，是一个统计变量。
- priority_：记录的是任务的优先级。

⑩ stack_unit_t 是否为有符号或无符号类型并不重要，关键是其所占字节的大小必须为 4。

■ stack_base_和 stack_size_：用于存放任务栈空间的开始地址及栈空间的字节大小。
■ entry_和 argument_：用于记录任务体函数及其参数。
■ name_：任务的名字被保存在这个变量中。

了解了任务数据结构后，接下来可以关注各任务控制函数的具体实现了。

20.4.1 任务创建

创建任务的函数是所有任务控制函数中最复杂的，因为它需要完成任务栈的创建和生成初始任务情景。

20.4.1.1 函数基本实现

任务是通过调用 task_create()函数创建的，该函数的实现如图 20.18 所示。在调用该函数前需要使用 STACK_DECLARE 宏为任务分配好栈空间。此外，为任务指定优先级和名字也是调用 task_create()函数时必须做的。task_create()函数返回 0 表示任务被成功地创建了，此时 _p_handle 参数指向新创建的任务。

```
00045: #define STACK_WIDTH_IN_BYTES     4
00046:
00058: static task_t g_task_pool [TASK_PRIORITY_LEVELS];
00065: static dll_t g_allocated_task;
00066:
00353: error_t task_create (task_handle_t *_p_handle, const char _name [],
00354:     task_priority_t _priority, stack_unit_t *_stack_base, usize_t _stack_bytes)
00355: {
00356:     task_handle_t handle;
00357:     interrupt_level_t level;
00358:     error_t ecode;
00359:
00360:     if (is_in_interrupt ()) {
00361:         return ERROR_T (ERROR_TASK_CREATE_INVCONTEXT);
00362:     }
00363:     if (_priority > TASK_LAST_INDEX) {
00364:         return ERROR_T (ERROR_TASK_CREATE_INVPRIO);
00365:     }
00366:
00367:     handle = &g_task_pool [_priority];
00368:
00369:     level = global_interrupt_disable ();
00370:     if (MAGIC_NUMBER_TASK == handle->magic_number_) {
00371:         ecode = ERROR_T (ERROR_TASK_CREATE_PRIOINUSE);
00372:         goto error;
00373:     }
00374:     memset (handle, 0, sizeof (*handle));
00375:     handle->magic_number_ = MAGIC_NUMBER_TASK;
00376:     global_interrupt_enable (level);
00377:
00378:     if (0 == _name) {
00379:         handle->name_ [0] = 0;
00380:     }
```

```
00381:     else {
00382:         strncpy (handle->name_, _name, (usize_t)sizeof (handle->name_) - 1);
00383:         handle->name_ [sizeof (handle->name_) - 1] = 0;
00384:     }
00385:     handle->timeout_ = 0;
00386:     handle->priority_ = _priority;
00387:     handle->stack_base_ = ((address_t)_stack_base +
00388:         (STACK_WIDTH_IN_BYTES - 1)) & (~(STACK_WIDTH_IN_BYTES - 1));
00389:     handle->stack_size_ = _stack_bytes - (handle->stack_base_ -
00390:         (address_t)_stack_base);
00391:     handle->stack_size_ &= ~(STACK_WIDTH_IN_BYTES - 1);
00392:
00393:     level = global_interrupt_disable ();
00394:     ecode = task_state_change (handle, TASK_STATE_CREATED);
00395:     if (0 != ecode) {
00396:         handle->magic_number_ = 0;
00397:         goto error;
00398:     }
00399:     dll_push_tail (&g_allocated_task, &handle->node_);
00400:     global_interrupt_enable (level);
00401:
00402:     *_p_handle = handle;
00403:     return 0;
00404:
00405: error:
00406:     global_interrupt_enable (level);
00407:     return ecode;
00408: }
```

图 20.18

第 58 行定义了 ClearRTOS 的任务数组 g_task_pool，数组中每个元素对应于一个任务。由于在 ClearRTOS 中一个优先级只能对应于一个任务，因此 ClearRTOS 所支持的优先级数就是整个系统所支持的任务数，这也正是为什么 g_task_pool 数组元素的个数定义为 TASK_PRIORITY_LEVELS 的原因。

第 360～362 行用于防止函数在中断服务程序中被调用。第 363～365 行检查用户所提供的优先级是否合法。第 367 行从任务数组中获得优先级所对应的任务管理数据结构。第 370 行通过检查 magic_number_是否与 MAGIC_NUMBER_TASK 宏相同，来判断任务是否已经被创建过。在第 374 行对任务管理数据结构进行置 0 初始化。第 375 行将 magic_number_设置成 MAGIC_NUMBER_TASK。由于 magic_number_被初始化后就可以防止出现竞争问题，因此在第 376 行恢复中断的状态。

第 378～390 行对任务管理数据结构中的任务名、超时等待值、优先级和栈空间进行初始化。在初始化栈空间时需要考虑边界对齐问题。

第 394 行调用 task_state_change()函数将任务的状态设置为“已创建”状态。如果状态设置不成功，则需要在第 396 行将 magic_number_变量置为 0，表示该任务并没有被成功分配出去；如果状态设置成功，第 399 行将已分配出来的任务放入 g_allocated_task 链表中。

task_state_change()函数主要完成三方面的工作——对任务情景和任务栈进行初始化，以及

为任务分配定时器，其实现细节可以从图 20.19 中找到。

```
00106: error_t task_state_change (task_handle_t _handle, task_state_t _new_state)
00107: {
00108:     error_t ecode = 0;
00109:
00110:     if (_handle->state_ == _new_state) {
00111:         return 0;
00112:     }
00113:
00114:     switch (_new_state)
00115:     {
00116:     case TASK_STATE_CREATED:
00117:     {
00118:         #define TIMER_PREFIX "Task:"
00119:         char timer_name [NAME_MAX_LENGTH + 1] = TIMER_PREFIX;
00120:
00121:         // initialize the stack with value of MAGIC_NUMBER_STACK
00122:         stack_unit_t *p_unit = (stack_unit_t *)_handle->stack_base_;
00123:         int count = _handle->stack_size_ >> STACK_WIDTH_SHIFT4BYTE;
00124:
00125:         while (count > 0) {
00126:             *p_unit ++ = MAGIC_NUMBER_STACK;
00127:             count --;
00128:         }
00129:         context_init (&_handle->context_, _handle->stack_base_,
00130:             _handle->stack_size_, (address_t) task_main);
00131:         strncpy (&timer_name [sizeof (TIMER_PREFIX)], _handle->name_,
00132:             sizeof (timer_name) - (sizeof (TIMER_PREFIX) + 1));
00133:         timer_name [sizeof (timer_name) - 1] = 0;
00134:         ecode = timer_alloc (&_handle->timer_, timer_name, TIMER_TYPE_INTERRUPT);
00135:         if (0 != ecode) {
00136:             return ecode;
00137:         }
00138:     }
00139:     break;
00140:     ……
00181:     _handle->state_ = _new_state;
00182:     return 0;
00183: }
```

图 20.19

第 123 行得到任务堆栈所占用的大小，以栈的最小操作单位（x86 处理器是 4 字节）来计算。第 125～128 行对栈空间进行初始化，将每一个栈单元用 MAGIC_NUMBER_STACK 宏进行赋值，这是为栈溢出检测做准备的，在 20.7 节有详述。第 129 和 130 行调用 context_init()函数，对任务的情景进行初始化。第 131～133 行为任务所需的定时器准备名字，第 134 行调用 timer_alloc()函数从定时器管理模块中分配一个定时器。完成任务转换后，在第 181 行记录下任务的新状态。

20.4.1.2　任务情景初始化

任务情景的初始化需要调用 context_init()函数来实现，其实现如图 20.20 所示。

platform/arch/x86/simulator/src/context.c

```
00031: #define CPU_STACK_ALIGNMENT     4
00032: #define BOTTOM_MAGIC_NUMBER     0xDEADDEAD
00033:
00037: typedef struct {
00038:     register_t ebx_, esi_, edi_, eip_;
00039:     task_context_t *p_context_;
00040:     register_t old_esp_;
00041: } root_frame_t;
00042:
00075: void context_init (task_context_t *_p_context,
00076:     address_t _stack_base,usize_t _stack_size, address_t _task_entry)
00077: {
00078: #if defined(__i386__)
00079:     address_t stack_high;
00080:     root_frame_t *p_frame;
00081:
00082:     stack_high = _stack_base + _stack_size;
00083:     stack_high &= ~(CPU_STACK_ALIGNMENT - 1);
00084:
00085:     p_frame = (root_frame_t *)(stack_high - sizeof (root_frame_t));
00086:     p_frame->ebx_ = (register_t) BOTTOM_MAGIC_NUMBER;
00087:     p_frame->esi_ = (register_t) BOTTOM_MAGIC_NUMBER;
00088:     p_frame->edi_ = (register_t) BOTTOM_MAGIC_NUMBER;
00089:     p_frame->eip_ = (register_t) _task_entry;
00090:     //lint -e(545)
00091:     p_frame->p_context_ = _p_context;
00092:
00093:     root_frame_init (p_frame);
00094: #else
00095: #error "Oops! Unsupported CPU!"
00096: #endif
00097: }
```

图 20.20

初始化任务情景之前，需要先设置好任务的栈。第 37～41 行定义了 root_frame_t 结构，用于表示任务的栈根（或栈底）。第 82～83 行对栈的边界进行对齐处理。第 85 行从任务的栈空间的底部分配出 root_frame_t 结构所需的空间。第 86～88 行分别将结构中的 ebx_、esi_、和 edi_变量进行初始化，这三个值将用于初始化任务运行时处理器的 EBX、ESI 和 EDI 寄存器的初值。第 89 行保存任务入口函数的地址，在 task_state_change()函数中可以看到，任务入口函数为 task_main()函数，该函数实现在 20.4.1.3 节有详述。第 91 行将保存任务情景的内存指针赋值给结构的 p_context_变量。第 93 行调用 root_frame_init()函数初始化任务栈根。

图 20.21 示例说明了调用 root_frame_init()函数之前栈空间的内存布局。其中示例说明了两块不同的栈空间，上面一块是待创建任务的栈空间，下面是任务创建函数调用者的栈空间，调用者可能是某一任务，抑或是 main()函数。注意：区分出存在两块不同的栈空间非常重要，但两块栈空间在地址空间内的上下关系并不重要。

root_frame_init()函数的实现可以从图 20.22 中找到，对应的汇编代码在图 20.23 中列出。由于 root_frame_init()是一个 C 函数，其被调用时一开始会创建自己的栈帧，这从图 20.23 中内存地址 40478c 和 40478d 处的汇编代码可以得到证实。执行完 40478f～404791 处的代码后，栈空间布局如图 20.24 所示。

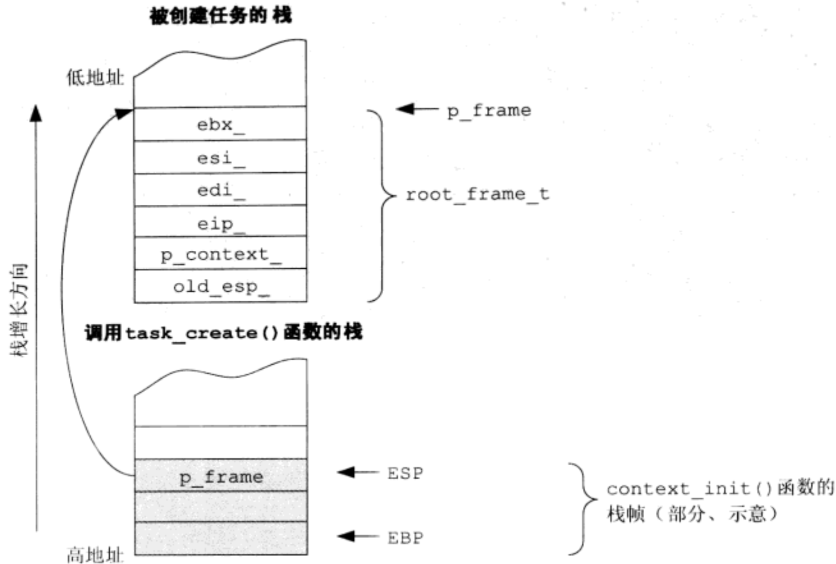


图 20.21

```
00053: static void root_frame_init (root_frame_t * _p_frame)
00054: {
00055: asm volatile(
00056:     // save old ESP into root_frame_t
00057:     "movl %%esp, 0x14(%0) \n"
00058:     // switch to new stack which starts from _p_frame
00059:     "movl %0, %%esp \n"
00060:     // initialize EBX
00061:     "popl %%ebx \n"
00062:     // initialize ESI
00063:     "popl %%esi \n"
00064:     // initialize EDI
00065:     "popl %%edi \n"
00066:     // The reason why we don't want to initialize the EBP is when the code is
00067:     // build with -O2 option, the GCC will optimize the code to use EBP at the end
00068:     // of root_context_init (). So if we initialize the EBP here it will cause crash.
00069:     "call _root_context_init \n"
00070:     "movl 8(%%esp), %%esp \n"
00071:     ::"r"(_p_frame):"ebx","esi","edi"
00072:     );
00073: }
```

图 20.22

```
                     /embedded/build
$ objdump -S -d debug/taskv1.exe
...... 显示结果有删减 ......
static void root_frame_init (root_frame_t * p_frame)
{
  40478c:       55                      push   ebp
  40478d:       89 e5                   mov    esp, ebp
  40478f:       57                      push   edi
  404790:       56                      push   esi
  404791:       53                      push   ebx
asm volatile(
  404792:       8b 45 08                mov    0x8(ebp), eax
  404795:       89 60 14                mov    esp,0x14(eax)
  404798:       89 c4                   mov    eax, esp
```

```
 40479a:     5b                        pop    ebx
 40479b:     5e                        pop    esi
 40479c:     5f                        pop    edi
 40479d:     e8 ce ff ff ff            call   404770 <context init>
 4047a2:     8b 64 24 08               mov    0x8(esp), esp
  "call  root context init \n"
  "movl 8(  esp),    esp   \n"
  ::"r"( p frame):"ebx","esi","edi"
  );
}
 4047a6:     5b                        pop    ebx
 4047a7:     5e                        pop    esi
 4047a8:     5f                        pop    edi
 4047a9:     5d                        pop    ebp
 4047aa:     c3                        ret
```

图 20.23

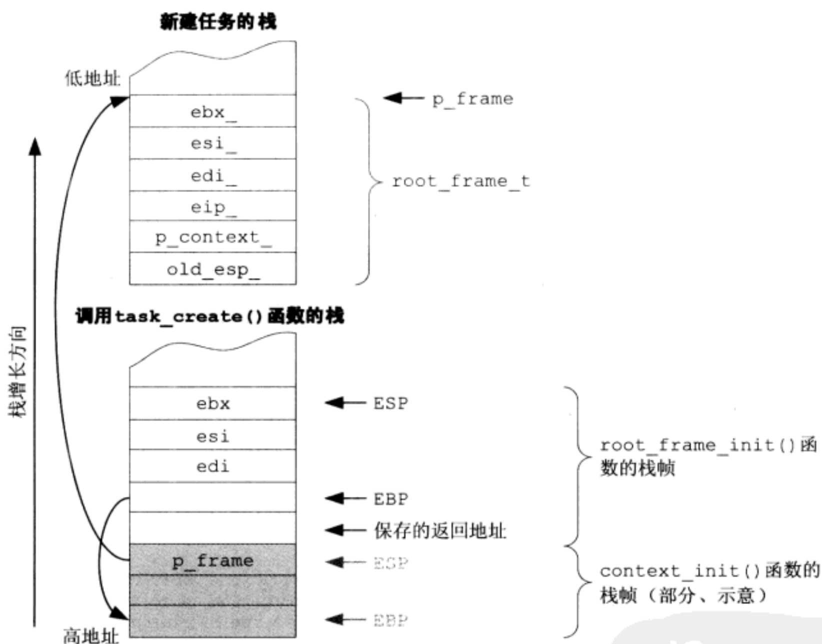


图 20.24

在图 20.22 的嵌入汇编代码中，“%0”表示的就是_p_frame 参数，从图 20.23 中的汇编代码可以看到，内存地址 404792 处的指令就是将_p_frame 参数放到 EAX 寄存器中。

图 20.22 中的第 57 行将 ESP 寄存器的值保存到栈根的 old_esp_变量中，第 59 行则将 p_frame 所指向的内存地址当做 ESP 寄存器的新值，这一步操作使处理器的 ESP 寄存器切换到了新建任务的栈空间，此时的栈空间布局如图 20.25 所示。请注意，此时 EBP 寄存器并没有指向新建任务的栈，因此在新建任务的栈上还没有形成栈帧。

接着通过第 61、63 和 65 行的三个 pop 退栈操作，将 p_frame 中对应变量的值加载到处理器的对应寄存器中，此时栈空间布局如图 20.26 所示。第 69 行通过调用 root_context_init()函数

将新任务的情景保存到内存中，该函数的实现列于图 20.27 中，图 20.28 是其汇编代码。

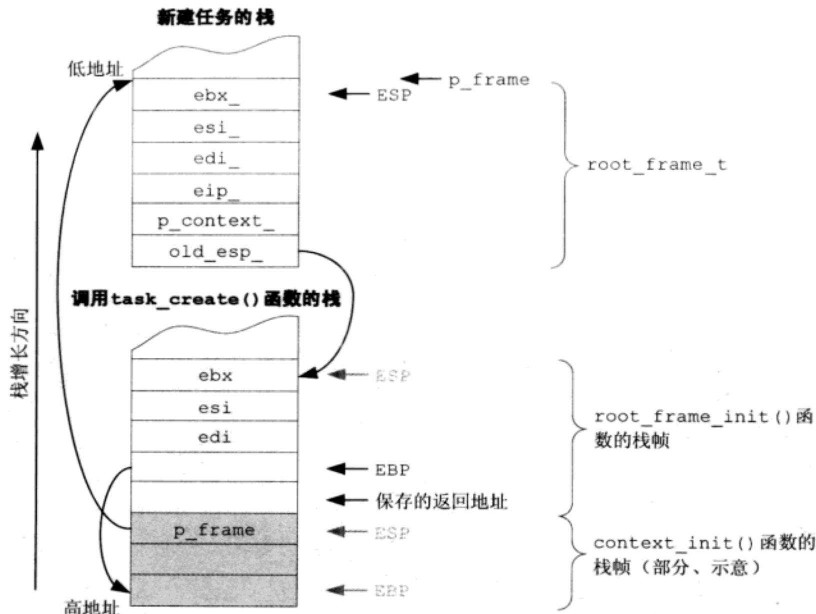


图 20.25

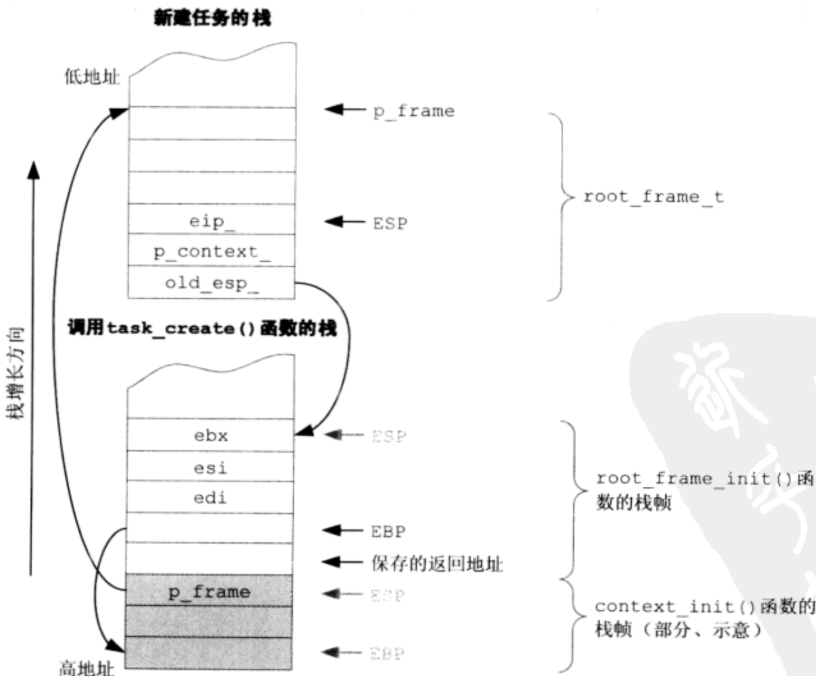


图 20.26

```
embedded/code/platform/arch/simulator/context.c
00045: static void root_context_init (void (*_task_entry)(), task_context_t *_p_context)
00046: {
00047:     if (context_save (_p_context)) {
00048:         _task_entry ();
00049:     }
00050: }
```

图 20.27

```
                    /embedded/build
$ objdump -S -d debug/taskv1.exe
...... 显示结果有删减 ......
static void root_context_init (void (*_task_entry)(), task_context_t *_p_context)
{
  404770:       55                      push   %ebp
  404771:       89 e5                   mov    %esp,%ebp
  404773:       83 ec 08                sub    $0x8,%esp
    if (context_save (_p_context)) {
  404776:       8b 45 0c                mov    0xc(%ebp),%eax
  404779:       89 04 24                mov    %eax,(%esp)
  40477c:       e8 c7 02 00 00          call   404a48 <_context_save>
  404781:       85 c0                   test   %eax,%eax
  404783:       74 05                   je     40478a <_root_context_init+0x1a>
        _task_entry ();
  404785:       8b 45 08                mov    0x8(%ebp),%eax
  404788:       ff d0                   call   *%eax
    }
}
  40478a:       c9                      leave
  40478b:       c3                      ret
```

图 20.28

显然，root_context_init()函数也需要构建自己的栈帧，栈帧形成后的内存布局如图 20.29 所示。

注意：这是第一次在新建任务的栈空间上形成了栈帧。在 10.4.2.1 节中指出，当参数是整型或者指针类型时，第一个参数的位置将位于 8(%ebp)处，第二个参数则位于 12(%ebp)处。图 20.29 所示的栈空间布局使得栈根中的 eip_和 p_context_两个变量成为了 root_context_init()函数所需的两个参数，这是非常有技巧的一处实现。

在 root_context_init()函数的实现中，将调用 context_save()函数为新任务保存情景，这部分内容与讲解 context_switch()函数时相同。当新创建的任务第一次被调度运行时，图 20.27 中的第 48 行将会被运行[11]。从上面的叙述我们可知，root_context_init()函数的第一个参数即为 root_frame_t 中的 eip_变量，而 eip_变量实际指向 task_main()函数。因此，当一个新建任务在创建后第一次被调度运行时，task_main()是它的第一个调用函数，这有点类似于 C 程序的入口函数总是 main()一样。

⑪ 请读者对照 context_switch()与 root_context_init()两个函数实现上的区别加以理解。

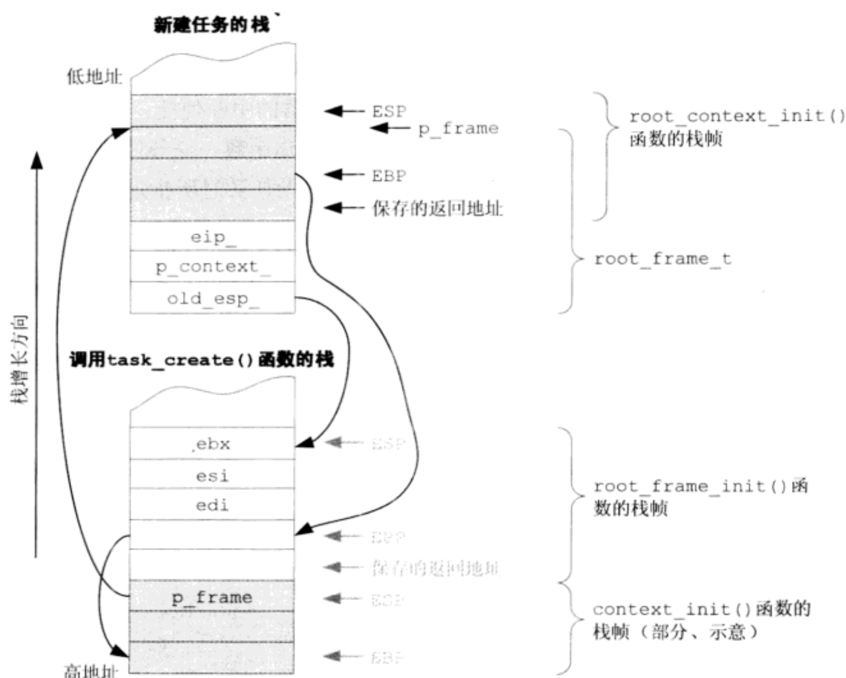


图 20.29

当 root_context_init()函数返回以后，又将获得图 20.26 所示的栈空间布局，且程序将回到 root_frame_init()函数中继续运行。图 20.22 中的第 70 行处将保存在 root_frame_t 结构中 old_esp_ 变量中的值赋值给 ESP 寄存器，其实就是回到了图 20.24 所示的栈空间布局。也就是说，完成了新建任务情景的初始化后，从新建任务的栈切回到调用 task_create()函数的栈上。在此之后的程序又都是 C 函数调用了，在此不再做更多的解释。

20.4.1.3　任务入口函数

task_main()是所有任务的入口函数，其实现如图 20.30 所示。

```
00078: static void task_main ()
00079: {
00080:     task_handle_t handle = g_task_running;
00081:
00082:     global_interrupt_enable (INTERRUPT_ENABLED);
00083:     // the entry function should not return, otherwise, it means the
00084:     // task is going to be exited
00085:     handle->entry_ (handle->name_, handle->argument_);
00086:     // delete the task if the entry function returns, task_delete ()
00087:     // could return an error code if this is the second time to delete
00088:     // it, so, always ignore return value
00089:     (void) task_delete (handle);
00090: }
```

图 20.30

task_main()函数也将以创建自己的栈帧作为运行后续程序的开始。task_schedule()函数调用context_switch()函数之前，g_task_running 会被初始化为指向将要运行的任务，图 20.30 中的第 80 行处通过该变量得到当前运行任务。在第 82 行先将处理器的中断使能。第 85 行则调用使用 task_start()函数（参见 20.4.2 节）启动任务时所指定的任务体函数。任务体函数的第一个参数是任务的名称，第二个参数同样是在调用 task_start()函数启动任务时所指定的。

一旦第 85 行的任务体函数返回说明任务即将被终止，在第 89 行 task_main()的最后，通过调用 task_delete()函数删除任务。在 task_delete()函数（参见 20.4.3 节）被调用时将触发一次任务切换，其结果就是被删除任务的 task_main()函数永远没有再次被运行的机会。从栈布局（参见图 20.29）来看，我们不允许 task_main()函数返回，否则会出现错误，这一点请读者思考为什么。

从 task_main()函数的实现来看，任务的终止有两种途径：一是通过调用 task_delete()函数显式中止；二是采用让任务体函数直接返回的方式。第二种方式类似于从 C 程序的 main()主函数中返回表示整个程序退出。

20.4.2　任务启动

任务创建好以后，还不会被调度器调度运行，而是需要对其进行一次启动操作，这正是 task_start()函数的功能。task_start()函数的实现如图 20.31 所示。

```
00448: error_t task_start (task_handle_t _handle, task_entry_t _entry, void *_p_arg)
00449: {
00450:     interrupt_level_t level;
00451:     error_t ecode;
00452:
00453:     level = global_interrupt_disable ();
00454:     if (is_invalid_handle (_handle)) {
00455:         ecode = ERROR_T (ERROR_TASK_START_INVHANDLE);
00456:         goto error;
00457:     }
00458:     if (TASK_STATE_CREATED != _handle->state_) {
00459:         ecode = ERROR_T (ERROR_TASK_START_INVOP);
00460:         goto error;
00461:     }
00462:     _handle->entry_ = _entry;
00463:     _handle->argument_ = _p_arg;
00464:     (void) task_state_change (_handle, TASK_STATE_READY);
00465:     global_interrupt_enable (level);
00466:
00467:     task_schedule (null);
00468:     return 0;
00469:
00470: error:
00471:     global_interrupt_enable (level);
00472:     return ecode;
00473: }
```

图 20.31

task_start()函数有三个参数。第一个参数是任务标识，用于指定所需启动的任务是哪一个；第二个参数是任务体函数的指针；第三个函数是传给任务体函数的第二个参数。

在第 454 行先检查任务句柄是否合法。第 458 行检查任务的状态是否是“已创建”。第 462 和 463 行分别保存任务体函数指针和传给任务体函数的参数。第 464 行调用 task_state_change() 函数，将任务的状态变为“就绪”。在第 467 行，一旦一个任务被启动了，就需要通过调用 task_schedule()函数尝试触发一次调度，这是为了确保最高优先级的任务被启动后能第一时间获得运行机会。

调用 task_state_change()函数将任务设置为“就绪”状态的程序实现如图 20.32 所示。

```
00107: error_t task_state_change (task_handle_t _handle, task_state_t _new_state)
00108: {
00109:     error_t ecode = 0;
00110:
00111:     if (_handle->state_ == _new_state) {
00112:         return 0;
00113:     }
00114:
00115:     switch (_new_state)
00116:     {
00117:     ......
00139:     case TASK_STATE_READY:
00140:         {
00141:             if (TASK_STATE_RUNNING == _handle->state_) {
00142:                 break;
00143:             }
00144:             if (timer_is_started (_handle->timer_)) {
00145:                 (void) timer_stop (_handle->timer_, null);
00146:             }
00147:             task_bitmap_bit_set (&g_ready_bitmap, _handle->priority_);
00148:             g_priority_map [_handle->priority_] = _handle;
00149:         }
00150:         break;
00151:     ......
00182: }
```

图 20.32

而从图 20.16 中可以看出，任务不只存在从“已创建”至“就绪”状态的迁移。当任务从“等待”状态迁移到“就绪”状态时需要停止已启动的定时器，这是第 144～146 行的作用。第 147 行在任务的就绪位图中设置任务优先级所对应的比特位，这样任务就有机会被调度器调度而获得运行机会了。第 148 行在优先级映射数组中设置优先级所对应的任务句柄，g_priority_map 数组的功用在下一章的 21.2.4 节还将涉及。

20.4.3　任务删除

删除任务需要调用 task_delete()函数，其实现列于图 20.33 中。

[illegible]/code/platform/task/v1/src/task.c

```
00410: error_t task_delete (task_handle_t _handle)
00411: {
00412:     interrupt_level_t level;
00413:     bool schedule_needed = false;
00414:     error_t ecode = 0;
00415:
00416:     if (is_in_interrupt () && STATE_UP == system_state ()) {
00417:         return ERROR_T (ERROR_TASK_DELETE_INVCONTEXT);
00418:     }
00419:     level = global_interrupt_disable ();
00420:     if (is_invalid_handle (_handle)) {
00421:         ecode = ERROR_T (ERROR_TASK_DELETE_INVHANDLE);
00422:         goto error;
00423:     }
00424:     ecode = task_state_change (_handle, TASK_STATE_DELETED);
00425:     if (0 != ecode) {
00426:         goto error;
00427:     }
00428:     _handle->magic_number_ = 0;
00429:     if (_handle == g_task_running) {
00430:         schedule_needed = true;
00431:     }
00432:     dll_remove (&g_allocated_task, &_handle->node_);
00433:     global_interrupt_enable (level);
00434:
00435:     // only task deleting itself needs a re-schedule, if a task is
00436:     // deleted by other running task, that's to say the running task's
00437:     // priority is higher than deleted one, so we don't need to do a re-schedule.
00438:     if (schedule_needed) {
00439:         task_schedule (null);
00440:     }
00441:     return 0;
00442:
00443: error:
00444:     global_interrupt_enable (level);
00445:     return ecode;
00446: }
```

图 20.33

对于该函数的实现，在此只指出其中的几个关键点。

- 在第 413 行定义的 schedule_needed 变量，用于指示任务的删除操作是否需要触发一次任务调度。在第 429 行，判断所删除的任务是否是正在运行的任务，如果是，则表示需要触发一次任务调度。其思路很简单，当前正在运行的任务一定是优先级最高的任务，如果它被删除则一定需要通过调度器找到系统中优先级最高的“就绪”任务。最后，在第 438 行通过检查 schedule_needed 变量来决定是否触发一次任务切换。
- 任务一旦被删除，其所对应的管理结构也就无效了。因此，在第 428 行将 magic_number_ 变量的值置为 0。
- 在第 432 行，需要将被删除的任务结点从链表 g_allocated_task 中移除。

当一个任务被删除时，需要清除就绪位图中的比特位，以及删除在任务创建时所分配的定

时器。这些工作是在 task_state_change()函数中完成的，其程序实现如图 20.34 所示。

```
embedded/code/platform/task/v1/src/task.c
00107: error_t task_state_change (task_handle_t _handle, task_state_t _new_state)
00108: {
00109:     error_t ecode = 0;
00110:
00111:     if (_handle->state_ == _new_state) {
00112:         return 0;
00113:     }
00114:
00115:     switch (_new_state)
00116:     {
00117: ……
00173:     case TASK_STATE_DELETED:
00174:         {
00175:             task_bitmap_bit_clear (&g_ready_bitmap, _handle->priority_);
00176:             (void) timer_free (_handle->timer_);
00177:         }
00178:         break;
00179:     ……
00182: }
```

图 20.34

20.4.4 任务挂起

通过调用 task_suspend()函数可以将任务挂起，其实现如图 20.35 所示。

```
embedded/code/platform/task/v1/src/task.c
00475: error_t task_suspend (task_handle_t _handle)
00476: {
00477:     interrupt_level_t level;
00478:     bool schedule_needed = false;
00479:     error_t ecode;
00480:
00481:     level = global_interrupt_disable ();
00482:     if (is_invalid_handle (_handle)) {
00483:         ecode = ERROR_T (ERROR_TASK_SUSPEND_INVHANDLE);
00484:         goto error;
00485:     }
00486:     if (TASK_STATE_CREATED == _handle->state_) {
00487:         ecode = ERROR_T (ERROR_TASK_SUSPEND_NOTSTARTED);
00488:         goto error;
00489:     }
00490:
00491:     ecode = task_state_change (_handle, TASK_STATE_SUSPENDING);
00492:     if (0 != ecode) {
00493:         goto error;
00494:     }
00495:     if (_handle == g_task_running) {
00496:         schedule_needed = true;
00497:     }
00498:     global_interrupt_enable (level);
00499:     // only task suspending itself needs a re-schedule, if a task is
00500:     // suspended by other running task, that's to say the running task's
00501:     // priority is higher than suspended one, so we don't need to do a re-schedule.
```

```
00502:     if (schedule_needed) {
00503:         task_schedule (null);
00504:     }
00505:     return 0;
00506:
00507: error:
00508:     global_interrupt_enable (level);
00509:     return ecode;
00510: }
```

图 20.35

task_suspend()函数的实现与前面介绍的 task_delete()函数很相似，只是调用 task_state_change()函数的目的是为了迁移到“挂起”状态。task_state_change()函数的实现如图 20.36 所示。

```
00107: error_t task_state_change (task_handle_t _handle, task_state_t _new_state)
00108: {
00109:     error_t ecode = 0;
00110:
00111:     if (_handle->state_ == _new_state) {
00112:         return 0;
00113:     }
00114:
00115:     switch (_new_state)
00116:     {
00117:     ......
00155:     case TASK_STATE_SUSPENDING:
00156:         {
00157:             if (timer_is_started (_handle->timer_)) {
00158:                 (void) timer_stop (_handle->timer_, &_handle->timeout_);
00159:             }
00160:             task_bitmap_bit_clear (&g_ready_bitmap, _handle->priority_);
00161:         }
00162:     ......
00182: }
```

图 20.36

任务被挂起时，如果被挂起的任务正处于等待状态，则需要停止等待定时器，并将定时器剩余的到期时间保存到任务句柄的 timeout_变量中，这是第 157～159 行代码的功用。在第 160 行将被挂起任务的优先级比特位从就绪位图中清除。

从实现来看，当一个任务被挂起时，如果被挂起的任务正处于等待状态，则被挂起的时间并不被算入等待时间中。这是从简化程序实现的角度做出的设计决定。

20.4.5 任务恢复

恢复一个被挂起的任务需要使用 task_resume()函数，其实现如图 20.37 所示。

```
00512: error_t task_resume (task_handle_t _handle)
00513: {
00514:     interrupt_level_t level;
```

```
00515:     error_t ecode;
00516:
00517:     level = global_interrupt_disable ();
00518:     if (is_invalid_handle (_handle)) {
00519:         ecode = ERROR_T (ERROR_TASK_RESUME_INVHANDLE);
00520:         goto error;
00521:     }
00522:     if (TASK_STATE_SUSPENDING != _handle->state_) {
00523:         ecode = ERROR_T (ERROR_TASK_RESUME_NOTSUSPENDED);
00524:         goto error;
00525:     }
00526:     if (0 == _handle->timeout_) {
00527:         (void) task_state_change (_handle, TASK_STATE_READY);
00528:     }
00529:     else {
00530:         (void) task_state_change (_handle, TASK_STATE_WAITING);
00531:     }
00532:     global_interrupt_enable (level);
00533:
00534:     task_schedule (null);
00535:     return 0;
00536:
00537: error:
00538:     global_interrupt_enable (level);
00539:     return ecode;
00540: }
```

图 20.37

在第 526～531 行根据 timeout_中的值是否为 0，以决定任务在恢复后应当进入哪一个状态。调用 task_state_change()函数设置任务为“就绪”状态的实现在前面已经讲过了，设置任务为“等待”状态的实现在下一节介绍。

20.4.6　任务睡眠

在 ClearRTOS 中，task_sleep()函数有两个作用：一，其参数不为 0 时，通过调用该函数使任务处于等待状态；二，其参数为 0 时，调用该函数，表示任务放弃一次运行机会。task_sleep()函数的实现如图 20.38 所示。

```
00548: error_t task_sleep (msecond_t _duration)
00549: {
00550:     task_handle_t handle = g_task_running;
00551:     interrupt_level_t level;
00552:     preschedule_callback_t callback = null;
00553:
00554:     if (is_in_interrupt ()) {
00555:         return ERROR_T (ERROR_TASK_SLEEP_INVCONTEXT);
00556:     }
00557:
00558:     if (0 == _duration) {
00559:         // Duration equals to 0, that means the task wants to yield the CPU.
00560:         callback = task_yeild_cpu;
00561:     }
00562:     level = global_interrupt_disable ();
```

```
00563:     handle->timeout_ = _duration;
00564:     (void) task_state_change (handle, TASK_STATE_WAITING);
00565:     global_interrupt_enable (level);
00566:
00567:     task_schedule (callback);
00568:     return 0;
00569: }
```

图 20.38

第 554～556 行用于防止调用是在中断状态发生的。第 558 行检查延时时间是否为 0，如果为 0，表示只是放弃当前所获得的处理器，因此在第 560 行将 task_shedule()函数参数设置为指向 task_yeild_cpu()函数的指针。第 563 行将睡眠时间记录到任务句柄中的 timeout_变量中，以便传入 task_state_change()函数内。第 564 行调用 task_state_change()函数将任务的状态变为“等待”。最后，在第 567 行调用 task_schedule()函数触发一次任务调度。

task_state_change()函数将任务设置为“等待”状态的程序实现如图 20.39 所示。

```
embedded/code/platform/task/v1/src/task.c
00107: error_t task_state_change (task_handle_t _handle, task_state_t _new_state)
00108: {
00109:     error_t ecode = 0;
00110:
00111:     if (_handle->state_ == _new_state) {
00112:         return 0;
00113:     }
00114:
00115:     switch (_new_state)
00116:     {
00117:     ……
00163:     case TASK_STATE_WAITING:
00164:         {
00165:             if (_handle->timeout_ != 0) {
00166:                 (void) timer_start (_handle->timer_,
00167:                     _handle->timeout_,task_timer_callback, _handle);
00168:                 _handle->timeout_ = 0;
00169:             }
00170:             task_bitmap_bit_clear (&g_ready_bitmap, _handle->priority_);
00171:         }
00172:         break;
00173:     ……
00182: }
```

图 20.39

第 165～169 行用于处理 timeout_变量不为 0 时的情形，此时应启动定时器进行等待。第 170 行清除就绪位图中的任务优先级位使任务不再参与调度。第 167 行所设置的定时器到期回调函数 task_timer_callback()的实现如图 20.40 所示。

```
00092: static void task_timer_callback (timer_handle_t _handle, void *_arg)
00093: {
00094:     task_handle_t p_task = (task_handle_t)_arg;
00095:     interrupt_level_t level;
00096:
```

```
00097:     UNUSED (_handle);
00098:
00099:     level = global_interrupt_disable ();
00100:     if (TASK_STATE_WAITING == p_task->state_) {
00101:         (void) task_state_change (p_task, TASK_STATE_READY);
00102:         p_task->ecode_ = ERROR_T (ERROR_TASK_WAIT_TIMEOUT);
00103:     }
00104:     global_interrupt_enable (level);
00105: }
```

图 20.40

在第 100 行验证任务是否仍处于“等待”状态。如果是，则在第 101 行调用 task_state_change()函数将任务的状态变为“就绪”，并在第 102 行将任务的返回值设置为等待超时，这个超时值在其他的模块中需要用到。当定时器到期时，并不需要在回调函数中触发一次任务调度，而是在其他地方有一个集中的触发点，在 20.8 节会就这一点做进一步的补充。

前面提到，当以参数 0 调用 task_sleep()函数时，指向 task_yeild_cpu()函数的指针将会被当做参数值传递给 task_schedule()函数，task_schedule()函数在调用 context_switch()函数之前会调用它。task_yeild_cpu()函数的实现示于图 20.41 中。

```
embedded/code/platform/task/v1/src/task.c
00542: static void task_yeild_cpu (task_handle_t _from, task_handle_t _to)
00543: {
00544:     UNUSED (_to);
00545:     (void) task_state_change (_from, TASK_STATE_READY);
00546: }
```

图 20.41

在第 545 行直接调用 task_state_change()函数，将调用 task_sleep()函数的任务设置为“就绪”状态。

回到图 20.38 中，当以参数 0 调用 task_sleep()函数时，同样会先将调用任务的状态设置为“等待”，接着调用 task_schedule()函数。在 task_schedule()函数中，会先计算出下一个将要运行的任务是哪一个。显然，调用 task_sleep()函数的任务不可能被调度器选中，因为它的状态并不是“就绪”。在 task_schedule()函数调用 context_switch()函数之前，会调用 task_yeild_cpu()回调函数，进而将调用 task_sleep()函数的任务状态又设置回“就绪”。这样下一次调度器选择运行任务时，又会将调用 task_sleep()函数的任务考虑在内了。最终的结果是，以参数 0 调用 task_sleep()函数使得调用任务放弃了一次运行机会。

20.5　竞争问题与中断控制

从应用程序的角度，对于竞争问题的理解一般是指如何使用互斥锁对共享资源进行保护。但是，从任务的角度并不能采用互斥锁这种方式，因为是先有任务后有互斥锁。

在操作系统的内核实现部分，需要用到另一种解决竞争问题的方法——中断控制。在本章

的前面部分，读者一定注意到了不少函数的实现中都用到了 global_interrupt_disable()和 global_interrupt_enable()两个函数，这两个函数的作用正是用来控制处理器的（可屏蔽）中断的。

20.5.1 竞争问题的产生

在讲解为什么需要使用控制中断的方法来解决竞争问题之前，我们需要先了解竞争问题是如何产生的。竞争问题的产生，说到底都是源于对共享资源的使用无法做到原子性。所谓原子性，就是对资源的存或取通过“一次”操作完成，“一次”是从处理器的指令角度出发的。对于图 20.42 中定义的 counter_increase()函数，如果这个函数会在中断状态（在中断服务程序中被调用）和非中断状态共同调用，则其中就存在竞争问题。

```
static int g_counter;

void counter_increase ()
{
   g_counter ++;
}
```

图 20.42

从处理器指令的角度来看，对于 g_counter 变量的加一操作，需要分成读、修改和写三个步骤来完成。

（1）从内存中将 g_counter 变量的值读入寄存器中。

（2）在寄存器中对值进行加一操作。

（3）将寄存器的值回写到 g_counter 变量所占用的内存中。

图 20.43 示例说明了当 counter_increase()函数在中断状态和非中断状态被“同时”调用时问题是如何发生的。这里的“同时”是指当非中断状态正在修改 g_counter 变量的值时发生了外部（硬件）中断，且中断服务程序又（直接或间接地）调用 counter_increase()函数。为了分析问题，图中假设一开始 g_counter 变量的值为 3，且 x86 处理器中的 EAX 寄存器被用于加工 g_counter 变量。

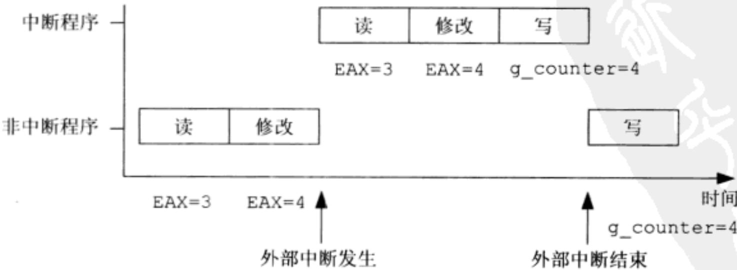


图 20.43

当 counter_increase()函数在非中断状态被调用时，因为 g_counter 变量的值为 3，所以 EAX 中的值从内存中读入后也为 3。接着，EAX 中的值被修改为 4。但是，修改过的值在还没有回写到 g_counter 变量所在内存中时中断发生了，这将中止非中断状态程序的运行，所有寄存器的值也将被压入栈中加以保存。

紧接着，counter_increase()函数在中断状态被调用，由于此时 g_counter 变量在内存中的值仍为 3，因此 counter_increase()函数在中断状态被调用完成后，g_counter 变量的值将被修改为 4。在中断结束时，保存在栈中的寄存器的值被恢复，也就是 EAX 的值将被恢复为 4。之后，在非中断状态下完成最后的内存写操作，使得 g_counter 变量中的值仍为 4。实际上，调用两次 counter_increase()函数后，g_counter 变量的值应为 5。正因为竞争问题的存在，使得出现了这种错误的结果。

20.5.2　通过中断控制解决竞争问题

为了解决这类竞争问题，需要考虑做到 g_counter 变量被修改时所需的三个步骤不会被中断打断，这可以通过开关中断的方式做到。处理器都会提供开关全局中断的控制寄存器。当中断被关闭时，所产生的外部硬件中断将不会被处理器处理，可以理解为中断被“挂起”了。一旦中断打开，已发生的中断将立即被处理器依据中断的优先级进行处理，即相应的中断服务程序会被调用。图 20.44 示例说明了如何通过中断控制这一方法来解决竞争问题。

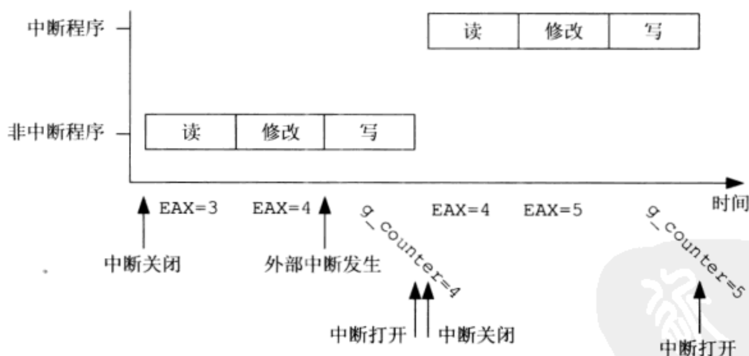


图 20.44

解决问题的方法是，每一次需要对 g_counter 变量进行加一操作之前，先关闭处理器的中断，而在完成了加一操作之后，又打开处理器的中断。更新后的 counter_increase()函数的实现如图 20.45 所示。

```
static int g_counter;

void counter_increase ()
{
```

```
   global_interrupt_disable ();
   g_counter ++;
   global_interrupt_enable ();
}
```

图 20.45

20.5.3 中断控制的嵌套问题

是不是在开、关中断的函数中只要对处理器的中断寄存器进行相应的设置就可以了呢？答案是否定的，因为这会造成另一个问题。

现在假设存在图 20.46 所示的函数，这个函数需要在它的实现中调用 counter_increase()函数，且 counter_increase()函数的实现为图 20.45 所示的方案。

```
example.c
void resource_alloc ()
{
   global_interrupt_disable ();
   ......
   counter_increase ();
   ......
   global_interrupt_enable ();
}
```

图 20.46

读者是否意识到了其中存在的问题：在 resource_alloc()函数调用了 counter_increase()函数之后，中断就被打开了。也就是说，在 resource_alloc()函数中，位于 counter_increase()函数之后的代码并没有消除竞争问题。导致问题的根源在于 global_interrupt_disable()和 global_interrupt_enable()函数的实现无法处理中断的嵌套控制。

要实现中断的嵌套控制，需要对中断控制函数做一定的修改。修改的思路是：在关闭中断时，global_interrupt_disable()函数返回之前的中断状态，该状态通过传给 global_interrupt_enable()函数用于恢复到之前的状态。图 20.47 示例说明了修改后的中断控制函数是如何被使用的。

```
static int g_counter;

void counter_increase ()
{
   interrupt_level_t level;

   level = global_interrupt_disable ();
   g_counter ++;
   global_interrupt_enable (level);
}

void resource_alloc ()
{
   interrupt_level_t level;
```

```
    level = global_interrupt_disable ();
    ......
    counter_increase ();
    ......
    global_interrupt_enable (level);
}
```

图 20.47

显然图中 level 必须是局部变量，否则嵌套问题依然存在。global_interrupt_enable()和 global_interrupt_disable()函数的实现将在 23.2.2 节做进一步介绍。

实际上，在大多的操作系统中，信号量、互斥锁等任务同步控制方法都是通过使用中断控制方法实现的[12]。

20.6　任务与中断状态

虽然处理器存在两种状态，即中断和非中断状态，但任务必须运行于非中断状态。因此，task_schedule()函数就不能在中断状态被调用。

实时操作系统为了做到对事件的快速响应，当处理器在退出中断状态时（即中断服务程序的最后面）可以实现任务切换。当任务模块在初始化时，会将 task_schedule_in_interrupt()函数作为中断处理结束时的回调函数向中断管理模块进行注册（参见 20.9 节），以便在中断结束时实现任务切换。图 20.48 示例说明了 task_schedule_in_interrupt()函数的实现。

```
00227: void task_schedule_in_interrupt ()
00228: {
00229:     interrupt_level_t level;
00230:     task_handle_t running, successor;
00231:     bool overflowed = false;
00232:
00233:     level = global_interrupt_disable ();
00234:     if (0 != g_scheduler_locked_count) {
00235:         global_interrupt_enable (level);
00236:         return;
00237:     }
00238:
00239:     successor = g_priority_map [task_bitmap_lowest_bit_get (&g_ready_bitmap)];
00240:     if (successor == g_task_running) {
00241:         global_interrupt_enable (level);
00242:         return;
00243:     }
00244:     running = g_task_running;
00245:     g_task_running = successor;
00246:     if (0 == is_stack_overflowed (running, &overflowed)) {
00247:         if (overflowed) {
00248:             //console_print ("\n\nError: stack overflow for task
```

[12] 有些处理器提供了原子操作指令，在这种情形下，信号量、互斥锁等同步控制方法就不需要通过开关中断的方式进行实现了。

```
00249:             // \"%s\"\n\n",running->name_);
00250:             (void) task_suspend (running);
00251:             g_statistics.overflowed_ ++;
00252:         }
00253:     }
00254:     if (TASK_STATE_RUNNING == running->state_) {
00255:         (void) task_state_change (running, TASK_STATE_READY);
00256:     }
00257:     (void) task_state_change (successor, TASK_STATE_RUNNING);
00258:     g_statistics.scheduled_ ++;
00259:     successor->stats_scheduled_ ++;
00260:     context_switch_in_interrupt (&running->context_, &successor->context_);
00261:     global_interrupt_enable (level);
00262: }
```

图 20.48

该函数的实现与 task_schedule()函数很类似，只是在需要进行任务切换时调用的是 context_switch_in_interrupt()函数而不是 context_switch()。由于目前 ClearRTOS 无法直接接管中断，因此 context_switch_in_interrupt()函数的实现是空的。

20.7　任务栈溢出检测

在 20.4.1 节中提及（图 20.19 的第 125～128 行）当一个任务被创建时，它的栈空间的每一个单元会被 task_state_change()函数初始化为 MAGIC_NUMBER_STACK 以用于实现任务栈的溢出检测。

当从一个任务切换到另一个任务时，调度函数就有检测被暂停的任务是否存在栈溢出的机会。图 20.49 示例说明了在 20.2.2 节中所省略的 task_schedule()函数的代码片段。

```
00184: void task_schedule (preschedule_callback_t _callback)
00185: {
00186:     interrupt_level_t level;
00187:     task_handle_t running, successor;
00188:     bool overflowed = false;
00189:     ......
00206:         if (0 == is_stack_overflowed (running, &overflowed)) {
00207:             if (overflowed) {
00208:                 console_print ("\n\nError: stack overflow detected for
00209:                     task \"%s\"\n\n",running->name_);
00210:                 (void) task_suspend (running);
00211:                 g_statistics.overflowed_ ++;
00212:             }
00213:         }
00214:     ......
00225: }
```

图 20.49

在 task_schedule()函数进行任务切换之前，会调用 is_stack_overflowed()函数（第 206 行）查看将被暂停的任务是否存在栈溢出现象。如果存在栈溢出问题，则在第 208 行输出错误日志，

并在第 210 行将存在栈溢出问题的任务挂起。第 211 行用于更新对应的统计信息。

栈溢出的检测原理很简单。当任务在创建时，给每一个栈单元设置初始值。栈空间的使用空间会随函数的调用深度而增加，一旦栈空间被使用过后，在初始化时所设置的初值就会被改变。检测的原理就是检查一个栈空间顶部的值是否仍为初始化时所设置的值，如果不是，则说明出现了栈溢出问题。图 20.50 示例说明了栈溢出检测函数 is_stack_overflowed()的实现。其中的第 659 行正是用于检测栈顶部的值是否仍为初始化时所设置的值，只比对两个栈单位的值是从性能方面考虑的。

```
00648: error_t is_stack_overflowed (const task_handle_t _handle, bool *_p_overflowed)
00649: {
00650:     interrupt_level_t level;
00651:     stack_unit_t *p_top;
00652:
00653:     level = global_interrupt_disable ();
00654:     if (is_invalid_handle (_handle)) {
00655:         global_interrupt_enable (level);
00656:         return ERROR_T (ERROR_TASK_STACK_INVHANDLE);
00657:     }
00658:     p_top = (stack_unit_t *)_handle->stack_base_;
00659:     if (p_top [0] != MAGIC_NUMBER_STACK || p_top [1] != MAGIC_NUMBER_STACK) {
00660:         *_p_overflowed = true;
00661:     }
00662:     else {
00663:         *_p_overflowed = false;
00664:     }
00665:     global_interrupt_enable (level);
00666:
00667:     return 0;
00668: }
```

图 20.50

除了可以了解栈空间是否存在溢出外，还可以得到任务栈的使用率。通过栈的使用率，可以帮助我们做出任务栈空间是否充足的判断。任务栈的使用率，通过调用 stack_used_percentage()函数来获取，图 20.51 是其具体实现。

```
00671: error_t stack_used_percentage (const task_handle_t _handle, int *_p_percentage)
00672: {
00673:     interrupt_level_t level;
00674:     stack_unit_t *p_top;
00675:     int nondirty_count = 0;
00676:
00677:     if (is_in_interrupt ()) {
00678:         return ERROR_T (ERROR_TASK_STACK_INVCONTEXT);
00679:     }
00680:
00681:     level = global_interrupt_disable ();
00682:     if (is_invalid_handle (_handle)) {
00683:         global_interrupt_enable (level);
00684:         return ERROR_T (ERROR_TASK_STACK_INVHANDLE);
00685:     }
```

```
00686:     p_top = (stack_unit_t *)_handle->stack_base_;
00687:     // we don't want to lock the interrupt to worsen the response for
00688:     // interrupt so we can lock the scheduler instead
00689:     scheduler_lock ();
00690:     global_interrupt_enable (level);
00691:
00692:     while (MAGIC_NUMBER_STACK == *p_top) {
00693:         nondirty_count ++;
00694:         p_top ++;
00695:     }
00696:     *_p_percentage = 100 -
00697:         nondirty_count*100/(int)(_handle->stack_size_ >> STACK_WIDTH_SHIFT4BYTE);
00698:
00699:     scheduler_unlock ();
00700:     return 0;
00701: }
```

图 20.51

由于这个函数可能会占用较多的处理器时间，所以不允许在中断状态下被调用（第 677 行）。第 689 行锁定调度器，第 690 行打开中断。之所以这样做，是因为对任务栈的扫描可能会因为栈空间较大造成耗时过长，从而造成中断的关闭时间过长，进而影响对中断的响应性。第 692～695 行从栈空间的顶部开始扫描，统计有多少栈单元中的值仍为初始化时的值。在第 696～697 行计算出栈空间的使用率。

20.8 滴答与空闲任务

每一个操作系统的实现都一定离不开一个概念——“滴答（tick）”。“滴答”一词源于老式时钟的钟摆在运动时所发出的滴答声。任务离不开延时等待这样的行为，比如本章介绍的 task_sleep()函数就是实现延时等待的方法。虽然在 task_sleep()的实现中，延时等待是通过软件定时器（参见第 24 章）实现的，但软件定时器是由谁去驱动的呢？正是通过“滴答”。

“滴答”通常是通过使用处理器的硬件定时器产生固定周期的中断。相邻两次中断之间的时间间隔可以根据需要加以配置，常见的有 10 毫秒或 60 毫秒。决定两次滴答之间的时间间隔需要权衡。间隔越短，每秒所产生的中断次数越多，定时器的精度将更高，但会因为频繁地中断处理器而影响性能；间隔越长，会降低定时器延时精度，但性能却更优。

实时操作系统通常能做到当滴答的中断处理完成时，在中断的返回过程中进行任务切换。由于目前的 ClearRTOS 并不能直接接管处理器的中断，因此无法做到这一点。正是因为 ClearRTOS 不存在中断这一问题，所以造成在保存它的任务情景时并不存在除了 ESP、EBP、EIP、ESI、EDI 和 EBX 之外的其他寄存器。如果问“为什么？”，在此不打算解释而是作为思考题留给读者。

尽管 ClearRTOS 不能接管处理器的中断，但还是需要通过一定的方式来实现“滴答”的功能。我们用 Linux 操作系统中的定时器信号（signal）来模拟“滴答”，具体实现请参见 23.4.1 节。在进一步说明 ClearRTOS 中“滴答”的处理之前，需要先了解空闲任务。

每一个操作系统都存在空闲任务的概念，当没有其他的任务需要运行时，空闲任务就会被运行。很容易想到，对于基于优先级调度的操作系统中空闲任务的优先级应是最低的。在真实的操作系统中，空闲任务可以理解为一个永远运行的死循环，但 ClearRTOS 中却不是这样。如果让空闲任务进行死循环，则会造成运行 ClearRTOS 的操作系统（Windows 或 Linux）变得很忙。

空闲任务的体函数 task_entry_idle()的实现如图 20.52 所示，它通过运用 select()函数等待信号。只要整个进程没有信号发生，空闲任务就会阻塞且不会占用处理器的时间。当滴答的信号产生时，会造成 select()函数返回 EINTR 错误码（第 46 行）。

```
00033: static bool g_tick_occurred_count;
00034: static statistic_t g_tick_delayed;
00035:
00036: void task_entry_idle (const char _name [], void *_p_arg)
00037: {
00038:     interrupt_level_t level;
00039:
00040:     UNUSED (_name);
00041:     UNUSED (_p_arg);
00042:
00043:     //lint -e{716}
00044:     while (1) {
00045:         if (select (0, 0, 0, 0, 0) < 0) {
00046:             if (errno != EINTR) {
00047:                 // should never happen
00048:                 console_print ("Fatal: non expected error occurred in"
00049:                     " task_entry_idle ()\n");
00050:             }
00051:
00052:             level = global_interrupt_disable ();
00053:             if (g_tick_occurred_count > 1) {
00054:                 g_tick_delayed ++;
00055:             }
00056:             while (g_tick_occurred_count > 0) {
00057:                 g_tick_occurred_count --;
00058:                 global_interrupt_enable (level);
00059:                 interrupt_enter ();
00060:                 timer_fire ();
00061:                 interrupt_exit ();
00062:                 level = global_interrupt_disable ();
00063:             }
00064:             global_interrupt_enable (level);
00065:
00066:             task_schedule (null);
00067:             continue;
00068:         }
00069:     }
00070: }
00084:
00085: statistic_t tick_delayed ()
00086: {
00087:     return g_tick_delayed;
00088: }
```

图 20.52

第 53～55 行是更新“滴答”被延期处理次数的统计信息。理论上，我们希望每一个滴答信号的出现都会得到及时的处理，通过这一统计信息有助于我们分析是否存在因为滴答延期处理而造成什么问题。第 56～63 行是根据所产生的滴答次数，调用相应次数的 timer_fire()函数（参见第 24 章）以驱动定时器管理模块对到期定时器的处理。当一个任务处于睡眠状态且到期时，timer_fire()函数的内部会最终调用任务定时器的回调函数 task_timer_callback()。在第 66 行通过调用 task_schedule()函数触发一次任务调度，因为有可能具有更高优先级的任务因为等待到期而进入了“就绪”状态，这也回答了为什么在 task_timer_callback()函数中不需要调用 task_schedule()函数触发任务调度这一问题。

空闲任务是在整个系统的初始化过程中通过调用 idle_task_spawn()函数创建的，idel_task_spawn()函数的实现如图 20.53 所示，它的实现也展示了在 ClearRTOS 中如何创建和启动一个任务。

```
00070: static task_handle_t g_task_idle;
00071:
00264: void idle_task_spawn ()
00265: {
00266:    STACK_DECLARE (stack, CONFIG_IDLE_TASK_STACK_SIZE);
00267:
00268:    (void) task_create (&g_task_idle, "Idle", IDLE_TASK_PRIORITY,
00269:        stack, sizeof (stack));
00270:    (void) task_start (g_task_idle, task_entry_idle, 0);
00271: }
```

图 20.53

第 70 行的 g_task_ide 全局变量用于保存空闲任务。第 266 行先定义空闲任务的栈空间内存，栈空间的大小由 CONFIG_IDLE_TASK_STACK_SIZE 宏表示。第 268 行调用 task_create()函数先创建任务，任务的优先级是 IDLE_TASK_PRIORITY，它被定义成 TASK_PRIORITY_LOWEST，是 ClearRTOS 中最低的优先级。最后在第 270 行调用 task_start()函数启动任务。

图 20.54 示说明例了“滴答”产生时的处理函数 tick_process()，它被当做回调函数注册到“滴答”设备（参见 23.4.1 节）中。

```
00076: void tick_process ()
00077: {
00078:    interrupt_level_t level;
00079:
00080:    level = global_interrupt_disable ();
00081:    g_tick_occurred_count ++;
00082:    global_interrupt_enable (level);
00083: }
```

图 20.54

“滴答”信号发生时的处理过程大致如下：

（1）“滴答”设备（参见 23.4.1 节）的中断服务程序被调用，其再调用 tick_process()函数。

（2）tick_process()函数对 g_tick_occurred_count 变量进行加一操作。

（3）“滴答”信号使得空闲任务所调用的 select()函数返回 EINTR，空闲任务根据 g_tick_occurred_count 变量的值调用相应次数的 timer_fire()函数。

20.9 多任务环境控制

ClearRTOS 中需要通过函数来控制是否进入多任务环境或从多任务环境中退出。进入多任务环境意味着调度器开始进行任务调度工作。调用 multitasking_start()函数使得进入多任务环境，退出多任务环境则需调用 multitasking_stop()函数，两个函数的实现如图 20.55 所示。

```
00072: static task_context_t g_start_context;
00073:
00074: static bool g_multitasking_started;
00075:
00300: void multitasking_start ()
00301: {
00302:     interrupt_level_t level;
00303:
00304:     level = global_interrupt_disable ();
00305:     if (g_multitasking_started) {
00306:         global_interrupt_enable (level);
00307:         return;
00308:     }
00309:     g_multitasking_started = true;
00310:     g_task_running = g_priority_map [task_bitmap_lowest_bit_get (&g_ready_bitmap)];
00311:     g_task_running->stats_scheduled_ ++;
00312:     g_statistics.scheduled_ ++;
00313:     interrupt_exit_callback_install (task_schedule_in_interrupt);
00314:     global_interrupt_enable (level);
00315:
00316:     // open and start the tick
00317:     if (device_open (&g_tick_handle, "/dev/clock/tick", 0) != 0) {
00318:         console_print ("Error: cannot open tick device");
00319:     }
00320:     if (device_control (g_tick_handle, OPTION_TICK_START,
00321:         (int)CONFIG_TICK_DURATION_IN_MSEC, tick_process) != 0) {
00322:         console_print ("Error: cannot start tick device");
00323:     }
00324:
00325:     context_switch (&g_start_context, &g_task_running->context_);
00326: }
00327:
00328: void multitasking_stop ()
00329: {
00330:     interrupt_level_t level;
00331:
00332:     level = global_interrupt_disable ();
```

```
00333:     if (!g_multitasking_started) {
00334:         global_interrupt_enable (level);
00335:         return;
00336:     }
00337:     g_multitasking_started = false;
00338:     global_interrupt_enable (level);
00339:
00340:     if (device_control (g_tick_handle, OPTION_TICK_STOP, 0, 0) != 0) {
00341:         console_print ("Error: cannot stop tick device");
00342:     }
00343:     if (device_close (g_tick_handle) != 0) {
00344:         console_print ("Error: cannot close tick device");
00345:     }
00346:
00347:     context_switch (&g_task_running->context_, &g_start_context);
00348: }
```

图 20.55

第 72 行的 g_start_context 变量用于记录多任务启动时的情景，以便在退出多任务环境时能返回。ClearRTOS 的现有实现中，多任务环境的启动是在 main()函数中完成的，因此 g_start_context 记录的是 main 主线程的情景。第 74 行的 g_multitasking_started 变量用于记录系统是否已启动多任务环境。

multitasking_start()函数在第 309 行将 g_multitasking_started 变量设置为 true，表示系统进入了多任务环境。第 310 行设置 g_task_running 变量以指向已创建任务中优先级最高的任务。第 311 和 312 行分别对相应的统计信息进行更新。第 317～323 行是打开“滴答”设备并启动它，“滴答”设备的驱动程序在 23.4.1 节进行讲解。在启动“滴答”设备时，设置的“滴答”处理回调函数是 tick_process()函数（第 321 行）。在第 325 行，通过调用 context_switch()函数使得最高优先级的任务被调度运行。

multitasking_start()函数启动多任务环境进行第一次调度时，并不能直接使用 task_switch()函数，因为 task_switch()函数的设计是基于两个任务间的（其中一个是被暂停的，另一个则是将要运行的），而 multitasking_start()函被调用时并没有任务处于运行状态。

multitasking_stop()函数除了将 g_multitasking_started 变量设置为 false 外（第 337 行），还得停止系统的“滴答”和关闭“滴答”设备（第 340～345 行），以及在最后通过调用 context_switch()函数切换回多任务环境的启动点（第 347 行）。

这里需要提醒一下读者：当 multitasking_start()函数被调用时这个函数并不会返回，因为最高优先级的任务会因为多任务环境的启动而运行，其返回是当 multitasking_stop()函数被调用之后才发生的。

20.10　任务模块管理

在第 14 章指出，每一个模块应为之设计一个模块回调函数，以实现模块的统一初始化和

终止化操作。图 20.56 是任务管理模块回调函数 module_task()的实现。

```
embedded/code/platform/task/v1/src/task.c
00273: static bool task_check_for_each (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00274: {
00275:     task_handle_t handle = (task_handle_t)_p_node;
00276:
00277:     UNUSED (_p_dll);
00278:     UNUSED (_p_arg);
00279:
00280:     console_print ("Error: task \"%s\" isn't deleted\n", handle->name_);
00281:     return true;
00282: }
00283:
00284: error_t module_task (system_state_t _state)
00285: {
00286:     if (STATE_INITIALIZING == _state) {
00287:         memset (&g_priority_map [0], 0, sizeof (g_priority_map));
00288:         task_bitmap_init (&g_ready_bitmap);
00289:         idle_task_spawn ();
00290:     }
00291:     else if (STATE_DESTROYING == _state) {
00292:         (void) task_delete (g_task_idle);
00293:         // check whether all tasks created have been deleted or not,
00294:         // if not take them as error
00295:         (void) dll_traverse (&g_allocated_task, task_check_for_each, 0);
00296:     }
00297:     return 0;
00298: }
```

图 20.56

任务模块只关注模块的初始化和终止化操作。当进行初始化时，需要对任务模块的相关全局变量进行初始化（第 287 和 288 行），以及创建空闲任务（第 289 行）。在模块被终止时，删除空闲任务（第 292 行），以及最后检查是否仍有任务没有被删除（第 295 行）。

在第 14 章中指出，各模块应当在终止化的过程中，检查所管理的资源是否已完全回收。第 295 行的目的就是检查在系统终止化时，是否所有的任务都被删除了。每一个被创建的任务都会被放入 g_allocated_task 链表中，通过使用 task_check_for_each()回调函数遍历这个链表，就可以知道哪些任务没有被删除。对于没有删除的任务，task_check_for_each()函数会在终端上打印一行错误信息（第 280 行）。

图 20.57 示例说明了用于查看任务模块信息和所有被创建任务状态的 task_dump()函数的实现。通过调用 task_dump()函数，将获得任务管理模块在.bss 段中所占用的大致内存，以及各任务的状态和相关的统计信息。

```
00736: static bool task_dump_for_each (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00737: {
00738:     task_handle_t handle = (task_handle_t)_p_node;
00739:     bool overflowed;
00740:     int percentage;
00741:
```

```
00742:     UNUSED (_p_dll);
00743:     UNUSED (_p_arg);
00744:
00745:     (void) is_stack_overflowed (handle, &overflowed);
00746:     (void) stack_used_percentage (handle, &percentage);
00747:     console_print (" Name: %s\n", handle->name_);
00748:     console_print (" Priority: %u\n", handle->priority_);
00749:     console_print (" Stack Base: %u\n", handle->stack_base_);
00750:     if (overflowed) {
00751:        console_print (" Stack Size: %u bytes (overflowed)\n",
00752:           handle->stack_size_);
00753:     }
00754:     else {
00755:        console_print (" Stack Size: %u bytes (%u%% used)\n",
00756:        handle->stack_size_, percentage);
00757:     }
00758:     console_print (" Scheduled: %u\n", handle->stats_scheduled_);
00759:     console_print (" State: %s\n", task_state_description (handle));
00760:     console_print ("\n");
00761:     return true;
00762: }
00763:
00764: void task_dump ()
00765: {
00766:     if (is_in_interrupt ()) {
00767:        return;
00768:     }
00769:
00770:     scheduler_lock ();
00771:     console_print ("\n\n");
00772:     console_print ("Summary\n");
00773:     console_print ("-------\n");
00774:     console_print (" Supported: %u\n", TASK_PRIORITY_LEVELS);
00775:     console_print (" Allocated: %u\n", dll_size (&g_allocated_task));
00776:     console_print (" .BSS Used: %u\n", ((address_t)&g_multitasking_started
00777:        - (address_t)g_task_pool) + sizeof (g_multitasking_started));
00778:     console_print ("\n");
00779:     console_print ("Statistics\n");
00780:     console_print ("----------\n");
00781:     console_print ("    Task Scheduled: %u\n", g_statistics.scheduled_);
00782:     console_print (" Stack Overflowed: %u\n", g_statistics.overflowed_);
00783:     console_print ("   Invalid Handle: %u\n", g_statistics.invalid_handle_);
00784:     console_print (" Tick Delayed: %u\n", tick_delayed ());
00785:     console_print ("\n");
00786:     console_print ("Task Details\n");
00787:     console_print ("------------\n");
00788:     (void) dll_traverse (&g_allocated_task, task_dump_for_each, 0);
00789:     console_print ("\n");
00790:     scheduler_unlock ();
00791: }
```

图 20.57

20.11 taskv1 示例程序

taskv1 示例程序的运行结果在本章的开始显示于图 20.1 中，这里我们介绍它的程序实现。

任何 C 程序都离不开定义并实现一个 main()函数，taskv1 的 main()函数在 main.h 文件中被定义，如图 20.58 所示。将 main()函数的实现放入 main.h 文件的目的是为了复用，因为 embedded 项目中很多示例程序都需要使用它。

```
00037: int module_registration_entry (int argc, char *argv []);
00038:
00039: int main (int argc, char *argv [])
00040: {
00041:     if (module_registration_entry (argc, argv) != 0) {
00042:         printf ("Error: module registration failure\n");
00043:         return -1;
00044:     }
00045:     printf ("\nSystem is going to be up\n");
00046:     if (0 != system_up ()) {
00047:         printf ("Error: system cannot be up\n");
00048:         return -1;
00049:     }
00050:     multitasking_start ();
00051:     printf ("\nSystem is going to be down\n");
00052:     system_down ();
00053:
00054:     return 0;
00055: }
```

图 20.58

第 37 行声明了 module_registration_entry()函数的原型，这个函数是每一个应用程序需要实现的，其中包含对各模块进行注册的代码。后面在查看 taskv1 示例程序的 main.c 文件时将看到它的具体内容。main()函数最先调用 module_registration_entry()函数实现对各模块的注册（第 41 行）。第 46 和 52 行的 system_up()和 sysmtem_down()读者一定不陌生，它们在第 14 章中已介绍过了。第 50 行用于启动多任务环境。

第 50 行对 multitasking_start()函数的调用将造成最高优先级的任务被运行，且在没有调用 multitasking_stop()函数的情形下并不会返回。一旦 multitasking_stop()函数被调用，multitasking_start()函数将从第 50 行返回，并继续后面的系统终止化操作。对照图 20.1 中 taskv1 程序的运行结果，读者一定能理出程序运行的脉络。

taskv1 示例程序的主体代码如图 20.59 所示。在这个示例程序中创建了三个任务，其中一个任务用于模拟栈溢出的情形。模拟栈溢出任务的体函数是 task_entry_overflow()，另外两个任务的任务体函数是 task_entry()。

为了节省篇幅，在本书的不少示例程序（仅限于示例程序）中没有对所有函数的返回值进行检查，这一点在现实项目中请不要模仿。相反，在现实项目中读者应养成关注函数返回值的编程好习惯。

```
00026: #include "main.h"
00027: #include "device.h"
```

```
00028: #include "console.h"
00029:
00030: static void task_entry_overflow (const char _name [], void *_p_arg)
00031: {
00032:     // create a BIG local variable for simulating stack overflow
00033:     int buffer [900];
00034:     memset (buffer, 0, sizeof (buffer));
00035:
00036:     UNUSED (_p_arg);
00037:
00038:     //lint -e{716}
00039:     while (1) {
00040:         console_print ("%s", _name);
00041:         (void) fflush (stdout);
00042:         (void) task_sleep (500);
00043:     }
00044: }
00045:
00046: static void task_entry (const char _name [], void *_p_arg)
00047: {
00048:     UNUSED (_p_arg);
00049:
00050:     //lint -e{716}
00051:     while (1) {
00052:         console_print ("%s", _name);
00053:         (void) fflush (stdout);
00054:         (void) task_sleep (500);
00055:     }
00056: }
00057:
00058: // pad is used for giving a cushion before g_stack_for_task0 and
00059: // g_stack_for_task1. Without this cushion the stack overflow of
00060: // task0will have side effect on other task(s).
00061: STACK_DECLARE (pad, 1024);
00062: STACK_DECLARE (g_stack_for_task0, 1024);
00063: STACK_DECLARE (g_stack_for_task1, 1024);
00064: STACK_DECLARE (g_stack_for_task2, 1024);
00065: static task_handle_t g_task0;
00066: static task_handle_t g_task1;
00067: static task_handle_t g_task2;
00068: static timer_handle_t g_timer;
00069: static device_handle_t g_ctrlc_handle;
00070:
00071: static void timer_callback (timer_handle_t _handle, void *_arg)
00072: {
00073:     UNUSED (_handle);
00074:     UNUSED (_arg);
00075:
00076:     task_dump ();
00077:     multitasking_stop ();
00078: }
00079:
00080: error_t module_testapp (system_state_t _state)
00081: {
00082:     // memset for pad is for preventing the OS from optimizing it out
00083:     memset (pad, 0, sizeof (pad));
00084:
00085:     if (STATE_INITIALIZING == _state) {
00086:         (void) task_create (&g_task0, "0", 11, g_stack_for_task0,
              sizeof (g_stack_for_task0));
```

```
00087:      (void) task_start (g_task0, task_entry_overflow, 0);
00088:
00089:      (void) task_create (&g_task1, "1", 16, g_stack_for_task1,
                sizeof (g_stack_for_task1));
00090:      (void) task_start (g_task1, task_entry, 0);
00091:
00092:      (void) task_create (&g_task2, "2", 19, g_stack_for_task2,
                sizeof (g_stack_for_task2));
00093:      (void) task_start (g_task2, task_entry, 0);
00094:
00095:      (void) timer_alloc (&g_timer, "Dump");
00096:      (void) timer_start (g_timer, 5000, timer_callback, 0);
00097:
00098:      (void) device_open (&g_ctrlc_handle, "/dev/ui/ctrlc", 0);
00099:  }
00100:  else if (STATE_DESTROYING == _state) {
00101:      (void) device_close (g_ctrlc_handle);
00102:      (void) timer_free (g_timer);
00103:      (void) task_delete (g_task2);
00104:      (void) task_delete (g_task1);
00105:      (void) task_delete (g_task0);
00106:  }
00107:  return 0;
00108: }
00109:
00110: int module_registration_entry (int argc, char *argv [])
00111: {
00112:  UNUSED (argc);
00113:  UNUSED (argv);
00114:
00115:  (void) module_register ("Interrupt", MODULE_INTERRUPT, CPU_LEVEL,
            module_interrupt);
00116:  (void) module_register ("Device", MODULE_DEVICE, DRIVER_LEVEL, module_device);
00117:  (void) module_register ("Timer", MODULE_TIMER, OS_LEVEL, module_timer);
00118:  (void) module_register ("Task", MODULE_TASK, OS_LEVEL, module_task);
00119:  (void) module_register ("TestApp", MODULE_TESTAPP, APPLICATION_LEVEL3,
            module_testapp);
00120:  return 0;
00121: }
```

图 20.59

第 46～56 行任务体函数的实现是将任务的名称打印出来，打印完后进行 500 毫秒的延时。第 30～44 行任务体函数的实现多了其中的第 33～34 行代码，这一部分代码的作用就是通过定义一个大局部变量（并对其进行初始化）来模拟栈溢出。第 110～121 行 module_registration_entry()函数的实现中注册了五个模块，前三个模块将在后续的章节中进行介绍，最后的 TestApp 模块就是 taskv1 所要实现的应用逻辑。第 62～64 行为三个任务分别定义了 1024 × 4 字节的栈空间。

第 80～108 行实现了 TestApp 模块的注册回调函数，其中分为初始化和终止化两个部分。第 86～93 行分别创建了三个任务，并指定不同的优先级。第 95～96 行分配一个时间间隔为 5 秒的定时器，定时器的回调函数实现位于第 71～78 行，它通过调用 task_dump()函数显示所有创建任务的状态，以及在最后调用 multitasking_stop()函数终止系统。

第 98 行则打开“/dev/ui/ctrlc”设备，这个设备提供了让我们在程序运行时通过按下键盘上的 Ctrl+C 组合键终止 taskv1 程序的运行。第 101～105 行以相反的顺序关闭“/dev/ui/ctrlc”设备、删除定时器和三个任务。“/dev/ui/ctrlc”设备的驱动程序将在 23.4.3 节中介绍。

第 61 行所定义的 pad 变量是必不可少的，因为任务 0 将模拟一个栈溢出问题，我们并不希望因为模拟栈溢出而造成其他的问题。增加 pad 变量的作用就是为栈溢出时提供额外的空间。请注意，pad 变量必须放在 g_stack_for_task0 变量之前。

如果注销图 20.59 中的第 105 行，编译并运行程序，将获得如图 20.60 所示的运行结果。

```
                     /embedded/build
$ make
                     /embedded/build
$ ./release/taskv1.exe

System is going to be up

Info: press Ctrl+C to terminate!
……显示结果有删减……
System is going to be down
Error: task "0" isn't deleted
Error: timer "Task:0" isn't deleted
```

图 20.60

从结果来看，由于在终止化时没有删除名称为“0”的任务，将造成在终端上打印出两条错误消息。第二条错误信息是因为任务创建时为之分配了一个定时器的缘故，任务的不删除使得该定时器也“泄漏”了。

20.12 任务钩子函数

钩子（hook）函数通俗地说就是回调函数，通过使用钩子函数能极大地提高被设计软件模块的灵活性和可定制性。在 ClearRTOS 中提供了三种与任务相关的钩子函数，这三种钩子函数分别针对任务创建、切换和删除三个时刻。图 20.61 示例说明了与任务钩子函数相关的头文件。

```
00032: typedef void (*task_create_hook_t) (task_handle_t _handle);
00033: typedef void (*task_switch_hook_t) (task_handle_t _from, task_handle_t _to);
00034: typedef void (*task_delete_hook_t) (task_handle_t _handle);
```

图 20.61

第 32～34 行定义了三类钩子函数的原型，除了针对任务切换有两个参数分别表示将要暂停和将要运行的任务外，其他两个函数都只有一个参数用于表示所创建或被删除的任务。图 20.62 是实现任务钩子函数的第一部分代码。

```
00029: #define CONFIG_MAX_TASK_CREATE_HOOK    8
00030: #define CONFIG_MAX_TASK_SWITCH_HOOK    8
```

```
00031: #define CONFIG_MAX_TASK_DELETE_HOOK    8
00032:
00033: #define TASK_CREATE_HOOK_LAST_INDEX    (CONFIG_MAX_TASK_CREATE_HOOK - 1)
00034: #define TASK_SWITCH_HOOK_LAST_INDEX    (CONFIG_MAX_TASK_SWITCH_HOOK - 1)
00035: #define TASK_DELETE_HOOK_LAST_INDEX    (CONFIG_MAX_TASK_DELETE_HOOK - 1)
00036:
00037: static task_create_hook_t g_create_table [CONFIG_MAX_TASK_CREATE_HOOK];
00038: static task_switch_hook_t g_switch_table [CONFIG_MAX_TASK_SWITCH_HOOK];
00039: static task_delete_hook_t g_delete_table [CONFIG_MAX_TASK_DELETE_HOOK];
00040:
00041: static bool hook_add (void *_table [], void *_hook, int _last_index)
00042: {
00043:     int idx = 0;
00044:
00045:     for (; idx <= _last_index; idx ++) {
00046:         if (_table [idx] != null) {
00047:             continue;
00048:         }
00049:         _table [idx] = _hook;
00050:         return true;
00051:     }
00052:     return false;
00053: }
00054:
00055: //lint -e{818}
00056: static bool hook_remove (void *_table [], void *_hook, int _last_index)
00057: {
00058:     int idx = 0;
00059:
00060:     for (; idx <= _last_index && _table [idx] != null; idx ++) {
00061:         if (_table [idx] == _hook) {
00062:             _table [idx] = null;
00063:             idx ++;
00064:             goto move;
00065:         }
00066:     }
00067:     return false;
00068:
00069: move:
00070:     // move all following hook(s) up
00071:     for (; idx <= _last_index && _table [idx] != null; idx ++) {
00072:         _table [idx - 1] = _table [idx];
00073:         _table [idx] = null;
00074:     }
00075:     return true;
00076: }
```

图 20.62

第 29～31 行分别定义了 ClearRTOS 所支持的三类任务钩子函数的最大数目，在这里都定义成了 8。第 37～39 行分别定义了三类钩子函数的存放数组。第 41～53 行定义了 hook_add() 函数，它的第一个参数将指向第 37～39 行所定义的三个数组；第二个参数则是指明所需增加的钩子函数；最后一个参数则指明了第一个参数中的钩子函数数组的最后一个索引值。hook_add()函数的实现是在第一个参数所指向的数组中找到一个空闲的元素（即为 null 的），然后将第二个参数所指向的钩子函数放入其中即可。

第 56～76 行是 hood_remove()函数的实现，它的行为很容易理解。除了从数组中找到所需删除的钩子函数的位置外，还将被删除位置之后的元素向前移位（第 71～74 行）。这是为了保证数组中所存放的钩子函数是连续的，以便在对数组进行遍历时一碰到为 null 的元素就知道后面没有钩子函数需要处理了。图 20.63 是三类钩子的增加与删除函数的实现。

embedded/code/platform/task/v2/src/hook.c

```
00078: error_t task_create_hook_add (task_create_hook_t _hook)
00079: {
00080:     interrupt_level_t level = global_interrupt_disable ();
00081:     if (!hook_add ((void **)g_create_table, _hook,
00082:         TASK_CREATE_HOOK_LAST_INDEX)) {
00083:         global_interrupt_enable (level);
00084:         return ERROR_T (ERROR_TASK_HOOK_CREATE_NOROOM);
00085:     }
00086:     global_interrupt_enable (level);
00087:     return 0;
00088: }
00089:
00090: error_t task_create_hook_remove (task_create_hook_t _hook)
00091: {
00092:     interrupt_level_t level = global_interrupt_disable ();
00093:     if (!hook_remove ((void **)g_create_table, _hook,
00094:         TASK_CREATE_HOOK_LAST_INDEX)) {
00095:         global_interrupt_enable (level);
00096:         return ERROR_T (ERROR_TASK_HOOK_CREATE_NOTFOUND);
00097:     }
00098:     global_interrupt_enable (level);
00099:     return 0;
00100: }
00101:
00113: error_t task_switch_hook_add (task_switch_hook_t _hook)
00114: {
00115:     interrupt_level_t level = global_interrupt_disable ();
00116:     if (!hook_add ((void **)g_switch_table, _hook,
00117:         TASK_SWITCH_HOOK_LAST_INDEX)) {
00118:         global_interrupt_enable (level);
00119:         return ERROR_T (ERROR_TASK_HOOK_SWITCH_NOROOM);
00120:     }
00121:     global_interrupt_enable (level);
00122:     return 0;
00123: }
00124:
00125: error_t task_switch_hook_remove (task_switch_hook_t _hook)
00126: {
00127:     interrupt_level_t level = global_interrupt_disable ();
00128:     if (!hook_remove ((void **)g_switch_table, _hook,
00129:         TASK_SWITCH_HOOK_LAST_INDEX)) {
00130:         global_interrupt_enable (level);
00131:         return ERROR_T (ERROR_TASK_HOOK_SWITCH_NOTFOUND);
00132:     }
00133:     global_interrupt_enable (level);
00134:     return 0;
00135: }
00136:
00148: error_t task_delete_hook_add (task_delete_hook_t _hook)
00149: {
00150:     interrupt_level_t level = global_interrupt_disable ();
```

```
00151:     if (!hook_add ((void **)g_delete_table, _hook,
00152:         TASK_DELETE_HOOK_LAST_INDEX)) {
00153:         global_interrupt_enable (level);
00154:         return ERROR_T (ERROR_TASK_HOOK_DELETE_NOROOM);
00155:     }
00156:     global_interrupt_enable (level);
00157:     return 0;
00158: }
00159:
00160: error_t task_delete_hook_remove (task_delete_hook_t _hook)
00161: {
00162:     interrupt_level_t level = global_interrupt_disable ();
00163:     if (!hook_remove ((void **)g_delete_table, _hook,
00164:         TASK_DELETE_HOOK_LAST_INDEX)) {
00165:         global_interrupt_enable (level);
00166:         return ERROR_T (ERROR_TASK_HOOK_DELETE_NOTFOUND);
00167:     }
00168:     global_interrupt_enable (level);
00169:     return 0;
00170: }
```

图 20.63

三类钩子操作函数的实现都很简洁，分别通过调用 hook_add()和 hook_remove()函数实现对相应类型钩子函数的增减操作。对钩子函数的真正调用，需要通过图 20.64 定义的三个函数。

```
code/platform/task/v2/src/hook.c
00103: void task_create_hook_traverse (task_handle_t _handle)
00104: {
00105:     int idx = 0;
00106:
00107:     for (; idx <= TASK_CREATE_HOOK_LAST_INDEX &&
00108:         g_create_table [idx] != null;idx ++) {
00109:         g_create_table [idx] (_handle);
00110:     }
00111: }
00104:
00138: void task_switch_hook_traverse (task_handle_t _from, task_handle_t _to)
00139: {
00140:     int idx = 0;
00141:
00142:     for (; idx <= TASK_SWITCH_HOOK_LAST_INDEX &&
00143:         g_switch_table [idx] != null;idx ++) {
00144:         g_switch_table [idx] (_from, _to);
00145:     }
00146: }
00147:
00173: void task_delete_hook_traverse (task_handle_t _handle)
00174: {
00175:     int idx = 0;
00176:
00177:     for (; idx <= TASK_DELETE_HOOK_LAST_INDEX &&
00178:         g_delete_table [idx] != null;idx ++) {
00179:         g_delete_table [idx] (_handle);
00180:     }
00181: }
```

图 20.64

对这三个函数的调用，依次应当放到 task_create()、task_schedule()和 task_delete()三个函数中，对 task.c 文件的变更可以从 embedded 项目的 code/platform/task/v2/src 目录中找到，在此不打算将它们列出。

20.13 任务变量

任务变量（task variable）在有的操作系统中又被称为线程本地存储（Thread Local Storage，TLS）。通过使用任务变量能大大地简化程序的设计，这是任务变量这个概念被引入的根本原因。本节将介绍任务变量的作用和实现原理。

20.13.1 taskv2 示例程序

首先通过 taskv2 示例程序来了解任务变量的作用，它的源程序如图 20.65 所示。

```
00033: static char *g_hello;
00034: static int g_count;
00035: static int g_total;
00036:
00037: static void task_entry (const char _name [], void *_p_arg)
00038: {
00039:     UNUSED (_p_arg);
00040:     //lint -e{740}
00041:     (void) task_variable_add ((value_t *)&g_hello);
00042:     //lint -e{740}
00043:     (void) task_variable_add ((value_t *)&g_count);
00044:
00045:     g_hello = malloc (32);
00046:     if (null == g_hello) {
00047:         console_print ("Error: malloc () failure\n");
00048:         return;
00049:     }
00050:     g_count = 5;
00051:
00052:     (void) snprintf (g_hello, 32, "Hello %s\n", _name);
00053:
00054:     while (g_count -- > 0) {
00055:         interrupt_level_t old_level;
00056:
00057:         console_print ("%u: %s", g_count, g_hello);
00058:         (void) fflush (stdout);
00059:
00060:         old_level = global_interrupt_disable ();
00061:         g_total ++;
00062:         global_interrupt_enable (old_level);
00063:
00064:         (void) task_sleep (300);
00065:     }
00066:
00067:     free (g_hello);
00068:     if (10 == g_total) {
00069:         // for shuting down the system
```

```
00070:         multitasking_stop ();
00071:     }
00072: }
00073:
00074: error_t module_testapp (system_state_t _state)
00075: {
00076:     static task_handle_t task0;
00077:     static task_handle_t task1;
00078:     static device_handle_t ctrlc_handle;
00079:     STACK_DECLARE (stack0, 1024);
00080:     STACK_DECLARE (stack1, 1024);
00081:
00082:     if (STATE_INITIALIZING == _state) {
00083:         (void) task_create (&task0, "Yun", 11, stack0, sizeof (stack0));
00084:         (void) task_start (task0, task_entry, 0);
00085:
00086:         (void) task_create (&task1, "Fang", 16, stack1, sizeof (stack1));
00087:         (void) task_start (task1, task_entry, 0);
00088:
00089:         (void) device_open (&ctrlc_handle, "/dev/ui/ctrlc", 0);
00090:     }
00091:     else if (STATE_DESTROYING == _state) {
00092:         (void) device_close (ctrlc_handle);
00093:         (void) task_delete (task1);
00094:         (void) task_delete (task0);
00095:     }
00096:     return 0;
00097: }
00098:
00099: int module_registration_entry (int argc, char *argv [])
00100: {
00101:     UNUSED (argc);
00102:     UNUSED (argv);
00103:
00104:     (void) module_register ("Interrupt", MODULE_INTERRUPT, CPU_LEVEL,
             module_interrupt);
00105:     (void) module_register ("Device", MODULE_DEVICE, DRIVER_LEVEL, module_device);
00106:     (void) module_register ("Timer", MODULE_TIMER, OS_LEVEL, module_timer);
00107:     (void) module_register ("Task", MODULE_TASK, OS_LEVEL, module_task);
00108:     (void) module_register ("TestApp", MODULE_TESTAPP, APPLICATION_LEVEL3,
             module_testapp);
00109:     return 0;
00110: }
```

图 20.65

taskv2 示例程序的源代码与 taskv1 大部分类似，不同点在于只创建了两个任务以及各任务的行为有所不同。两个任务的任务体函数都是第 37 行的 task_entry()，下面让我们关注该函数的实现。

第 33 和 34 行定义了两个变量，在第 41 和 43 行分别将这两个变量置为任务变量。第 45 行为 g_hello 变量申请内存，第 52 行初始化该内存。第 50 行将 g_count 变量初始化为 5，表示每一个任务将打印出 5 行信息。第 54～65 行将变量的信息输出到终端上。第 61 行对总的打印行数进行计数，当两个任务各打印了 5 行后，将调用 multitasking_stop()函数终止整个系统（第 68～71 行）。第 67 行将前面分配获得的内存进行释放。taskv2 示例程序的运行结果如图 20.66 所示。

```
                        /embedded/build
$ make
                        /embedded/build
$ ./release/taskv2.exe

System is going to be up

Info: press Ctrl-C to terminate!
4: Hello Yun
4: Hello Fang
3: Hello Yun
3: Hello Fang
2: Hello Yun
2: Hello Fang
1: Hello Yun
1: Hello Fang
0: Hello Yun
0: Hello Fang

System is going to be down
```

图 20.66

从运行结果来看，被定义为全局变量的 g_hello 和 g_count，对于“Yun”和“Fang”两个任务却有着不同的值且互不影响。这种效果正是通过任务变量做到的。

如果不使用任务变量，则“Yun”和“Fang”两个任务需要定义各自的类似于 g_hello 和 g_count 的变量，即需要定义四个全局变量，且任务体函数也需要独立编写，不能共用⑬。或者共用同一个任务体函数，但在任务体函数的实现中使用额外的 if 语句，以保证存取与任务相对应的变量。很明显，任务变量的使用大大地简化了代码和提高了复用性。

20.13.2 原理

我们将通过图 20.67 来理解任务变量的实现原理，图中以 taskv2 示例程序中的 g_count 变量为例。

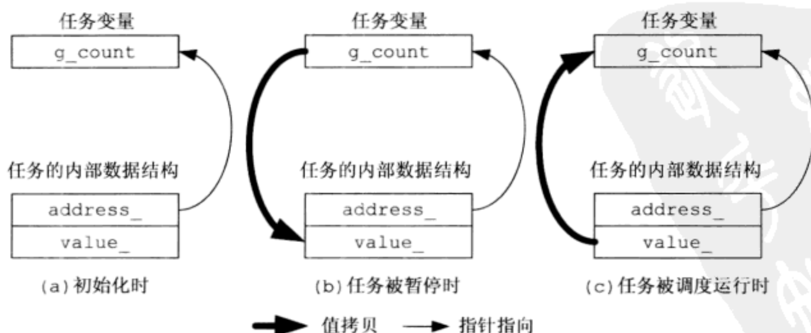


图 20.67

⑬ 这里假设应用场景需要使用全局变量，如果可以使用局部变量的话就无须采用任务变量作为实现方法。

当一个变量需要成为任务变量时，需要调用 task_variable_add()函数，且将变量的地址作为该调用函数的参数。task_variable_add()函数会为变量分配一个内部管理数据结构。内部数据结构中的 address_变量记录下被加入变量的地址，而 value_用于记录任务变量的值，即图中的 address_将保存 g_count 变量的地址，而 value_将保存 g_count 变量的值。

当任务发生切换时，如果任务是将被暂停的，g_count 中的值被放入内部数据结构的 value_变量中进行保存。反之，在任务被调度运行时，内部数据结构中 value_变量的值就赋值给 g_count 变量。对任务变量的保存和恢复，都是由任务变量管理模块在后台完成的。

20.13.3　实现

由于任务变量的保存与恢复需要在任务切换时完成，所以我们可以通过运用前面介绍的任务切换钩子函数来实现其功能。图 20.68 是任务变量模块的数据结构定义与初始化函数的实现代码。

```
00032: typedef struct {
00033:     dll_node_t node_;
00034:     address_t address_;
00035:     value_t value_;
00036: } task_variable_node_t;
00037:
```

```
00031: #define CONFIG_MAX_TASK_VARIABLE   32
00032: #define TASK_VARIABLE_LAST_INDEX   (CONFIG_MAX_TASK_VARIABLE - 1)
00033:
00034: static task_variable_node_t g_variable_pool [CONFIG_MAX_TASK_VARIABLE];
00035: static dll_t g_variable_free;
00036:
00067: static error_t task_variable_init ()
00068: {
00069:     int idx = 0;
00070:
00071:     for (; idx <= TASK_VARIABLE_LAST_INDEX; idx ++) {
00072:         dll_push_tail (&g_variable_free, &g_variable_pool [idx].node_);
00073:     }
00074:     return task_switch_hook_add (task_variable_switch_hook);
00075: }
```

图 20.68

先关注 taskvar.h 文件。第 32～36 行定义了用于管理任务变量的数据结构。其中的 node_起链表结点的功能，address_变量用于保存任务变量所在的内存地址，而 value_变量用于保存任务变量的值。

再看 taskvar.c 文件。第 34 行定义了用于管理任务变量的数组。第 35 行定义了用于存放没有分配出去的管理数据的链表。第 67～75 行定义了 task_variable_init()函数。第 71～73 行将数组中的每一个元素放入到空闲链表中。第 74 行调用 task_switch_hook_add()函数安装一个任务切换钩子函数。图 20.69 中可以找到钩子函数的实现。

```
00065: struct type_task {
00066:     ......
00086:     dll_t variable_;
00087: };
```

```
00037: static bool task_variable_store (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00038: {
00039:     task_variable_node_t *p_node = (task_variable_node_t *)_p_node;
00040:     value_t *p_value = (value_t *)p_node->address_;
00041:
00042:     UNUSED (_p_dll);
00043:     UNUSED (_p_arg);
00044:
00045:     p_node->value_ = *p_value;
00046:     return true;
00047: }
00048:
00049: static bool task_variable_restore (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00050: {
00051:     task_variable_node_t *p_node = (task_variable_node_t *)_p_node;
00052:     value_t *p_value = (value_t *)p_node->address_;
00053:
00054:     UNUSED (_p_dll);
00055:     UNUSED (_p_arg);
00056:
00057:     *p_value = p_node->value_;
00058:     return true;
00059: }
00060:
00061: static void task_variable_switch_hook (task_handle_t _from, task_handle_t _to)
00062: {
00063:     (void) dll_traverse (&_from->variable_, task_variable_store, 0);
00064:     (void) dll_traverse (&_to->variable_, task_variable_restore, 0);
00065: }
```

图 20.69

为了支持任务变量，需要对任务管理数据结构进行更改，即增加task.h中第86行的variable_链表变量，用于存放管理任务变量的内部数据结构，后面讲解task_variable_add()函数的实现时还将谈到这个新增变量。

当进行任务切换时，task_variable_switch_hook()函数会被调用。这个函数先把需要暂停任务的任务变量保存到任务句柄中（第63行），然后恢复将要运行任务的任务变量（第64行）。不论是保存任务变量还是恢复任务变量，都是对任务句柄中variable_链表中每一个结点进行遍历。当然，遍历时的操作有所不同，不同点分别体现在task_variabel_store()和task_variable_restore()两个函数的实现上，请读者自行理解。

增加一个任务变量需要调用task_variable_add()函数，该函数的实现如图20.70所示。

```
00077: error_t task_variable_add (value_t *_p_value)
00078: {
00079:     interrupt_level_t level;
```

```
00080:     static bool initialized = false;
00081:     task_variable_node_t *p_node;
00082:     task_handle_t handle = task_self ();
00083:
00084:     level = global_interrupt_disable ();
00085:     if (is_invalid_task (handle) || is_in_interrupt ()) {
00086:         global_interrupt_enable (level);
00087:         return ERROR_T (ERROR_TASK_VARIABLE_ADD_INVTASK);
00088:     }
00089:     if (!initialized) {
00090:         error_t ecode = task_variable_init ();
00091:         if (0 != ecode) {
00092:             global_interrupt_enable (level);
00093:             return ecode;
00094:         }
00095:         initialized = true;
00096:     }
00097:
00098:     p_node = (task_variable_node_t *)dll_pop_head (&g_variable_free);
00099:     if (null == p_node) {
00100:         global_interrupt_enable (level);
00101:         return ERROR_T (ERROR_TASK_VARIABLE_ADD_NOVAR);
00102:     }
00103:     p_node->address_ = (address_t)_p_value;
00104:     dll_push_tail (&handle->variable_, &p_node->node_);
00105:     global_interrupt_enable (level);
00106:     return 0;
00107: }
```

图 20.70

第 85～88 行是防止 task_variable_add()函数不是在任务的上下文和在中断状态下被调用。第 89～96 行检查任务变量管理模块是否被初始化过，如果没有则调用 task_variable_init()函数进行初始化。第 98 行从空闲链表中获取一个管理数据结构的实体，如果没有可用的实体，则在第 101 行返回对应的错误码。第 103 行保存任务变量的地址，第 104 行将管理实体放入任务的链表内。

现在只剩下 task_variable_remove()函数没有介绍了，其实现如图 20.71 所示。当需要删除一个任务变量时需要调用该函数。

```
00109: static bool task_variable_find (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00110: {
00111:     task_variable_node_t *p_node = (task_variable_node_t *)_p_node;
00112:
00113:     UNUSED (_p_dll);
00114:
00115:     if (p_node->address_ == (address_t)_p_arg) {
00116:         return false;
00117:     }
00118:     return true;
00119: }
00120:
00121: error_t task_variable_remove (value_t *_p_value)
00122: {
00123:     interrupt_level_t level;
```

```
00124:     task_variable_node_t *p_node;
00125:     task_handle_t handle = task_self ();
00126:
00127:     level = global_interrupt_disable ();
00128:     if (is_invalid_task (handle) || is_in_interrupt ()) {
00129:         global_interrupt_enable (level);
00130:         return ERROR_T (ERROR_TASK_VARIABLE_REMOVE_INVTASK);
00131:     }
00132:     p_node = (task_variable_node_t *)dll_traverse (
00133:         &handle->variable_,task_variable_find, _p_value);
00134:     if (null == p_node) {
00135:         global_interrupt_enable (level);
00136:         return ERROR_T (ERROR_TASK_VARIABLE_REMOVE_NOTFOUND);
00137:     }
00138:     dll_remove (&handle->variable_, &p_node->node_);
00139:     dll_push_tail (&g_variable_free, &p_node->node_);
00140:     global_interrupt_enable (level);
00141:     return 0;
00142: }
```

图 20.71

第 132 行先从任务句柄的链表中找到需要删除的任务变量所对应的管理数据（第 132～137 行），一旦找到则将管理实体从任务链表中删除（第 138 行），并放入空闲链表中（第 139 行）。

20.14 其他概念与思考

20.14.1 抢占式任务与实时系统的关系

实时操作系统（Real-Time Operating System，RTOS）中的“实时”，强调的是当某个事件发生时能在固定的时间内得到响应，即对事件的响应具有确定性。

为了实现对事件响应的确定性，实时操作系统的任务必须设计成具有“抢占式（pre-emptive）”的特点。抢占式的意思是：当高优先级任务所希望获得的资源一旦可用就让其“抢占”其他正在运行的任务而获得运行机会。下一章 21.2.4 节中的 provinv 和 provinh 两个示例程序能很好地让读者理解抢占式的含义。

操作系统所需响应的事件包含两大类：中断和程序内部产生的消息。中断的响应速度是实时操作系统很重要的一个指标，它是指当一个中断产生到运行中断服务程序的第一条指令之间的时间间隔。在 23.2.3 节中指出，为了跟踪处理器是否处于中断状态或非中断状态，中断发生时需要调用 interrupt_enter()函数，这个函数的调用时间应算入中断响应时间之内，因为它并不属于中断服务程序的一部分。

实时操作系统对于程序内部所产生的事件是通过隐式函数调用的形式加以响应的。比如，当一个任务正在等待一个被另一个任务所持有的互斥锁时，锁的持有者完成了锁的使用所进行的解锁操作其实隐含了对触发任务调度函数的调用。任务对程序内部所产生事件的响应速度可以理解为非中断状态下任务的切换速度。

一个嵌入式系统的实时性并不完全取决于操作系统，而是与应用程序的设计也有着紧密的关系。在 24.6 节以定时器管理模块为例讲解了如何通过设计来提高系统的实时性。

20.14.2　影响任务切换效率的因素

如何高效地完成任务切换是每一个实时操作系统在设计时的关键考虑因素。第一个影响因素与任务情景切换有关。前面看到，ClearRTOS 并没有将浮点寄存器纳入任务情景当中，对于具有浮点运算的操作系统往往要考虑浮点寄存器的保存与恢复。为了提高情景的切换效率，操作系统在设计时会为任务创建函数提供一个额外的参数，这个参数指明所创建的任务是否存在浮点运算。如果参数指明任务不存在浮点运算，则为该任务进行情景切换时就可以省去对浮点寄存器的保存与恢复操作。

第二个影响因素与调度算法有关，即如何从多个“就绪”的任务中快速地找到应获得处理器的任务。调度算法的效率对于实时操作系统的实时性起着决定性的作用。

第三个影响因素与任务切换钩子函数有关。比如，为了实现任务变量而增加的钩子函数就会影响任务的切换速度。显然，任务变量用得越多，对切换效率的影响就越大。因此，从软件开发的角度，在存在其他方案的情形下应尽量避免使用任务变量，当然也得少使用任务切换钩子函数。

20.14.3　避免直接删除任务

“直接删除任务”又可以称其为“简单粗暴地删除任务”。“直接”在这里所表达的意思是：在没有分析是否会造成副作用的情形下盲目地删除任务。

在 ClearRTOS 中，如果任务不是自己调用 task_delete()函数或从任务体函数中返回，那就存在被粗暴删除的嫌疑。

为什么说直接删除任务很危险呢？这是因为任务有可能在被删除之前获取了某种资源，而直接删除没有给任务释放这些资源的机会，这会给整个系统的正常运行带来极大的不确定性。打个比方，如果一个任务在被删除之前已经获得了一个互斥锁，直接删除的结果就是该锁将处于永远锁定的状态，这将导致锁所保护的资源永远不可用。再比如，如果一个任务在被删除之时正在写文件，这将导致获得一个不完整的文件。

为了避免直接删除任务所带来的危害，我们需要在设计时考虑如何终止一个任务。图 20.72 示例说明了一种实现。尽管这个实现简单化了，但是在我们的现实中一定能找到相类似的方案。

```
static semaphore_t g_semaphore;
static bool g_destroy_required;

void task_destroy ()
{
```

```
    g_destroy_required = true;
    semaphore_give (g_semaphore);
}

void task_entry (const char _name [], void *_p_arg)
{
    ......
    while (1) {
        semaphore_take (g_semaphore)
        if (g_destroy_required) {
            // free resources holden
            ......
            // delete myself
            task_delete (task_self ());
        }
        // do other matters
        ......
    }
}
```

图 20.72

这一方案中的 g_semaphore 信号量（参见 21.1 节）是用于通知任务有事务要处理的，而应用层所实现的 task_destroy()函数就利用了它。task_destroy()函数在给任务发送信号之前，先将 g_destroy_required 变量置为 true，以表示需要任务自行退出。当任务收到信号后，先检查 g_destroy_required 变量是否被置为 true，如果是则先释放所有已经获取到的资源，接着调用 task_delete()函数删除自己。

20.14.4 小心多任务设计被滥用

通过合理创建任务（或线程）的方式，可以有效地提高软件设计的模块性。另外，多任务在不少情形下，将提高系统的运行效率，因为一个任务在等待所需资源时，另一个任务可以利用处理器做其他的事。尽管多任务有它的好处，但把握度很重要。根据作者的观察，多任务设计方法大有被滥用之势，有的工程师习惯于一做设计就想到运用多任务。出现这种状况，是因为没有意识到多任务设计可能带来的问题。

多任务设计一定需要使用到任务同步方法（参见第 21 章），以保证多个任务有序地协同工作。但是，任务同步方法的使用并不是每个人都很擅长，即使觉得擅长也很容易一糊涂就设计出存在竞争问题的代码。再则，对于大型项目，由于代码量的急剧增长，多任务所带来的竞争问题更加不容易被发现，一旦发生问题就相当严重，而且不容易查错。任务数量使用得过多将导致更多的任务切换。也因为任务过多，而使得任务之间的通信所花费的处理器开销更大，并有可能造成系统性能问题。

作者也经历了从大量使用多任务设计到回归避免使用多任务的成长历程，也明白在不少情形下采用多任务设计是源于卖弄自己具备多任务的编程能力，以及愧疚于不采用多任务会造成系统性能问题。其实，一旦我们冷静地思考会发现，自认为多任务所带来的好处，在系统中很可能并不是关键。

从用户的体验来看，一款软件产品必须具备良好的鲁棒性[14]；否则，无论产品具有多好的功能特性最终都将被用户给抛弃。因此，软件在开发活动中的主旨之一应是采用容易获得高质量的设计方法，而不是运用更多的“高科技”。采用这种策略，允许我们适当地降低对团队技能的要求。

作者曾在一个不存在大负荷数据通信的项目中开发了一个运行于 Linux 操作系统之上、基于 TCP 套接字的网络通信框架，采用的是单线程的设计思想。这个框架通过采用 select()函数可以处理多个套接字的建链和通信。在设计的过程中，很强烈地体会到了采用单线程方式所带来的好处——程序更简单，调试更容易。

当我们考虑运用多任务设计时，静下心来思考一下它所带来的利与弊。一旦考虑清楚了应当采用多任务设计，那还是应当果断选择。

20.15　小结

掌握多任务实现原理的关键是理解任务情景。任务调度其实就是以一定的原则进行任务情景切换，其中很重要的是完成任务栈帧和程序计数器的切换。

通过为任务管理模块设计成提供安装钩子函数的功能将大大提高模块的灵活性，任务变量正是通过运用任务切换钩子函数来实现的。

我们应避免直接删除任务这种“粗暴”的行为，且提防多任务设计被滥用。

练习与思考

1. 在 20.1 节中介绍任务情景时，指出目前 ClearRTOS 的情景中只需要保存 ESP、EBP、EIP、ESI、EDI 和 EBX 这几个寄存器。在 20.8 节中介绍“滴答”时也指出，正是因为目前的 ClearRTOS 无法接管处理器的中断，从而造成很多寄存器不需要保存到任务情景中。这是为什么？

2. 当 task_schedule()在切换任务时，能否通过任务的栈寄存器 ESP 的值与任务栈的开始地址进行比较来判断是否存在栈溢出现象？

3. 在图 20.48 中，is_in_interrupt()函数的实现在比较 g_interrupt_nested_count 变量是否为 0 时需要关闭中断吗？

4. 既然 task_delete()函数会因为滥用而造成潜在的问题，那可否考虑在设计任务管理模块

[14] 鲁棒性(robustness)一词来源于自动控制领域，它是指一个控制系统在受到外部扰动的情形下仍能自动地克服扰动而恢复到稳定状态。软件行业借用这个词以表达所设计软件的容错能力。

时不提供 task_delete()函数？这一问题带给我们怎样的设计启示？

5. 当一个任务被直接删除时，它所持有的互斥锁将永远处于锁定状态，可否考虑采用一种设计，即当任务被删除时，将它所持有的锁进行自动释放？你如何看待这种设计思想？

6. 通过怎样的设计，能获得每个任务所占用处理器时间的百分比？



第 21 章

任务同步与通信，实现协同工作

在一个多任务的嵌入式系统中，要实现任务间的协同工作一定离不开同步与通信。同步这一术语很容易让人误解，以为它的目的是为了做到所有任务“同时”运行。同步的真实含义，是为了做到各任务“串行”运行以防止竞争问题的出现。

在嵌入式系统中，任务间同步与通信的方法主要有信号量、互斥锁、事件和消息队列。下面我们看一看每种机制的用途和在 ClearRTOS 中的实现。

21.1 信号量

每种同步方法都有它特定的应用场合。通过从应用场合入手，能更好地理解方法被提出的目的，进而掌握如何运用。

21.1.1 应用场合

假设有一个可容纳 300 辆车的地下车库，在出入口安装有一套控制设备。当有车想进入地下车库时，如果有空位则控制设备立即准许车进入，否则该车必须等待，直到有车离开车库。假设车库入口是由控制设备程序中的入口任务进行管理控制的，那如何实现该任务呢？

入口任务先要获得有车辆进入的信号，然后检查是否有空车位，如果有空车位则将阻挡杆升起以便车进入车库；如果没有空车位，任务应当被阻塞以等待空车位的出现。信号量（semaphore）对于车位这种同一资源有多种实例的场合就非常适合，图 21.1 示例说明了运用信号量来实现对车辆进入车库的管理。

```
semaphore_handle_t g_parking_lot_semaphore;

static void task_entrance (const char _name [], void *_p_arg)
{
    semaphore_create (&g_parking_lot_semaphore, "Parking Lot", 300);

    for (;;) {
```

```
        // 1) wait event of car enters
        ......
        // 2) get a parking lot
        semaphore_take (g_parking_lot_semaphore, WAIT_FOREVER);
        // 3) raise the stopping bar
        ......
    }
}

static void car_leaving_interrupt_handler (int _vector)
{
    ......
    semaphore_give (g_parking_lot_semaphore);
    ......
}
```

图 21.1

图中的 g_parking_lot_semaphore 变量是用于保存控制车位的信号量的，信号量需通过调用 semaphore_create()函数获得。调用 semaphore_create()函数时第三个参数设置为 300，表示一开始停车场有 300 个车位可用。task_entrance()函数调用 semaphore_take()函数以获得一个车位。如果信号量所记录的车位数大于 0，semaphore_take()会立即返回，且信号量所记录的车位数减一；当信号量所记录的车位数为 0 时，semaphore_take()函数将造成任务被阻塞，直到有车位可用才返回。

每当有车离开车库时，图中的 car_leaving_interrupt_handler()中断服务程序就会被调用，该中断服务程序通过调用 semaphore_give()函数对信号量进行加一操作，表示车库中空出来了一个车位。如果 task_entrance()函数（的调用任务）因为等待空闲车位处于阻塞状态，则 semaphore_give()函数的调用将造成其被唤醒而继续运行。

由此看来，信号量可以理解为是一个对同一资源进行计数的“计数器”。在这里的例子中“资源”所代表的是“可用车位”。semaphore_take()代表的是“拿”资源，其将造成“计数器”的值减一。如果所需的资源不可用，即“计数器”的值已为 0，将造成调用任务被阻塞，阻塞的方式可以是一直等到资源可用或者设置一个超时值以等待有限的时间。semaphore_give()代表的是“还”资源，其将造成“计数器”的值加一，这个函数永远不会造成阻塞。

我们对于信号量的应用场合做个总结。

- 信号量充当的是一个资源“计数器”。调用 semaphore_take()函数表示获取一个资源并将计数值减一。semaphore_give()函数的被调用则表示返还或提供资源，并将资源计数值加一。
- semaphore_give()函数可以在中断状态和任务环境中被调用，而 semaphore_take()只能在任务环境中被调用。这一点使得信号量除了可以作为任务之间的通信方法外，还可以作为中断与任务之间的通信手段。
- 信号量能实现任务之间的同步体现在两方面。一是当资源不够用时，调用 semaphore_take()函数的任务将被阻塞，直到资源可用或出现等待超时任务才会退出阻塞状态以继续运行。如果存在多个任务阻塞在同一个信号量上，任务的唤醒顺序会基于信号量的算法有序进行，目前 ClearRTOS 中是根据任务优先级的（优先级高的先唤

醒）。二是当多个任务需要对同一个信号量进行操作时，信号量的设计实现将保证各任务以串行的方式使用自己，从而解决多任务对信号量的竞争问题。

- 使用同一信号量的任务之间体现的是生产者和消费者关系。生产者通过调用 semaphore_give()函数告知消费者资源可用，而消费者通过调用 semaphore_take()函数获取资源。

接下来，让我们看一看 ClearRTOS 中的信号量是如何实现的。

21.1.2　程序实现

信号量与后面将要介绍的互斥锁在实现上有不少相似之处，在它们两者之间可以抽象出公共的概念以实现代码复用，这一抽象概念被本书称为同步对象（synchronization object）。在设计上我们通过使用回调函数，来实现信号量与互斥锁的差异部分。

21.1.2.1　同步对象的实现

同步对象及容器（container）的数据结构定义如图 21.2 所示。引入容器的概念，是为了将信号量与互斥锁分类管理，每一个容器只能存放一种类型的同步对象。

```
00033: typedef struct type_sync_object sync_object_t, *sync_object_handle_t;
00034:
00035: typedef bool (*sync_point_enter_t) (sync_object_handle_t _handle);
00036: typedef void (*sync_point_wait_t) (sync_object_handle_t _handle);
00037: typedef bool (*sync_point_exit_t) (sync_object_handle_t _handle,
       error_t *_p_ecode);
00038: typedef void (*sync_point_wake_t) (sync_object_handle_t _handle);
00039:
00040: typedef struct {
00041:     sync_point_enter_t enter_;
00042:     sync_point_wait_t wait_;
00043:     sync_point_exit_t exit_;
00044:     sync_point_wake_t wake_;
00045: } sync_operation_t, *sync_operation_handle_t;
00046:
00047: typedef struct {
00048:     magic_number_t magic_number_;
00049:     dll_t free_;
00050:     dll_t used_;
00051:     sync_operation_t opt_;
00052:     // statistics
00053:     statistic_t stats_noobj_;
00054: } sync_container_t, *sync_container_handle_t;
00055:
00056: struct type_sync_object {
00057:     dll_node_t node_;
00058:     magic_number_t magic_number_;
00059:     task_bitmap_t pending_bitmap_;
00060:     sync_container_handle_t container_;
00061:     char name_ [NAME_MAX_LENGTH + 1];
00062: };
```

图 21.2

第 56～62 行是同步对象数据结构的定义。每个同步对象是通过链表的形式存放在对应的容器中的，node_变量的作用就是用做链表结点。magic_number_用于标识一个同步对象是否有效。每一个同步对象中包含一个类型为 task_bitmap_t 的 pending_bitmap_变量，用于记录同步对象上有哪些任务正处于等待状态——等待获取同步对象所代表或保护的资源。container_变量指向的是同步对象的所属容器。name_表示同步对象的名称。第 33 行通过 typedef 将 type_sync_object 结构定义为 sync_object_t 类型。

容器数据结构的定义位于第 47～54 行。同样地，magic_number_变量用于标识该容器是否有效。free_和 used_两个链表分别用于存放空闲的和已分配出去的同步对象。addr_start_和 addr_end_两个变量记录了容器中的同步对象在内存中的开始和结束地址。后面将看到，不同类型的同步对象所占用的内存空间是以数组的形式定义的，因此是连续的内存空间。opt_变量中记录的是每一类同步对象的回调函数。最后，stats_noobj_用于统计容器内同步对象不足这一错误出现的次数。

一个同步对象所控制的其实是一个同步点[①]（synchronization point），对于一个同步点存在进入和离开两个动作。当进入一个同步点时，如果资源不可用则任务需要进入等待状态；当离开一个同步点时，则需要唤醒处于等待状态的其他任务。这正是定义第 40～45 行的 sync_operation_t 数据结构的指导思想。

1．容器初始化

同步对象容器的初始化是通过调用 sync_container_init()函数来完成的，其实现如图 21.3 所示。

```
00033: #define MAGIC_NUMBER_SYNCOBJ        0x53594E43L
00034: #define MAGIC_NUMBER_CONTAINER      0x4D414749L
00035:
00036: #define is_invalid_handle(_handle) \
00037:     ((_handle == null) || ((_handle)->magic_number_ != MAGIC_NUMBER_SYNCOBJ))
00038:
00039: //lint -e{818}
00040: error_t sync_container_init (sync_container_handle_t _handle, void *_p_objects,
00041:     usize_t _obj_count, usize_t _obj_size, const sync_operation_handle_t _opt)
00042: {
00043:     interrupt_level_t level;
00044:     usize_t idx = 0;
00045:     char *p_objects = (char *)(_p_objects);
00046:     sync_object_handle_t p_object;
00047:
00048:     if (is_in_interrupt ()) {
00049:         return ERROR_T (ERROR_SYNC_INIT_INVCONTEXT);
00050:     }
00051:
00052:     level = global_interrupt_disable ();
00053:     if (MAGIC_NUMBER_CONTAINER == _handle->magic_number_) {
00054:         global_interrupt_enable (level);
```

① 同步点这个术语是站在同步对象的角度而定义的。其粒度可大可小。

```
00055:         return ERROR_T (ERROR_SYNC_INIT_INVINIT);
00056:     }
00057:     memset (_handle, 0, sizeof (*_handle));
00058:     _handle->magic_number_ = MAGIC_NUMBER_CONTAINER;
00059:     global_interrupt_enable (level);
00060:
00061:     dll_init (&_handle->free_);
00062:     dll_init (&_handle->used_);
00063:     _handle->opt_ = *_opt;
00064:     memset (_p_objects, 0, _obj_count * _obj_size);
00065:     for (; idx < _obj_count; idx ++, p_objects += _obj_size) {
00066:         //lint -e{826}
00067:         p_object = (sync_object_handle_t) p_objects;
00068:         dll_push_tail (&_handle->free_, &p_object->node_);
00069:     }
00070:     return 0;
00071: }
```

图 21.3

sync_container_init()函数的第一个参数代表容器，由调用者传入；后面的三个参数分别是同步对象的开始地址、对象数量和一个同步对象所占内存的大小；最后一个参数指明容器中同步对象的回调函数。

第 48～50 行用于防止函数在中断状态下被调用。第 53～56 行检查容器是否已初始化过。第 57 行对容器的管理数据结构进行置 0 初始化。第 58 行对 magic_number_变量赋值，表示对应的容器已被初始化。第 53～58 行的操作是置于关闭中断保护之下的，用以保证容器的初始化不会出现竞争问题。第 61 和 62 行分别对容器中的两个链表进行初始化。第 63 行保存容器内同步对象的回调函数。第 64 行对所有的同步对象内存进行置 0 初始化。第 65～69 行将每一个同步对象放入空闲链表中。

2. 分配同步对象

当创建信号量或互斥锁时，需要从相应的容器中分配同步对象，这时需要调用 sync_object_alloc()函数，该函数的实现如图 21.4 所示。

```
00073: error_t sync_object_alloc (sync_container_handle_t _container,
00074:     sync_object_handle_t *_p_handle, const char _name [])
00075: {
00076:     interrupt_level_t level;
00077:     sync_object_handle_t handle;
00078:
00079:     if (is_in_interrupt ()) {
00080:         return ERROR_T (ERROR_SYNC_ALLOC_INVCONTEXT);
00081:     }
00082:
00083:     *_p_handle = null;
00084:     level = global_interrupt_disable ();
00085:     handle = (sync_object_handle_t) dll_pop_head (&_container->free_);
00086:     if (null == handle) {
00087:         *_p_handle = null;
00088:         _container->stats_noobj_ ++;
00089:         global_interrupt_enable (level);
```

```
00090:         return ERROR_T (ERROR_SYNC_ALLOC_NOOBJ);
00091:     }
00092:     memset (handle, 0, sizeof (*handle));
00093:     task_bitmap_init (&handle->pending_bitmap_);
00094:     if (0 == _name) {
00095:         handle->name_ [0] = 0;
00096:     }
00097:     else {
00098:         strncpy (handle->name_, _name, (usize_t)sizeof (handle->name_) - 1);
00099:         handle->name_ [sizeof (handle->name_) - 1] = 0;
00100:     }
00101:     handle->container_ = _container;
00102:     handle->magic_number_ = MAGIC_NUMBER_SYNCOBJ;
00103:     dll_push_tail (&_container->used_, &handle->node_);
00104:     global_interrupt_enable (level);
00105:     *_p_handle = handle;
00106:     return 0;
00107: }
```

图 21.4

第 85～91 行从容器的空闲链表中获取一个同步对象。如果没有同步对象可用，则在第 88 行更新统计信息并在第 90 行返回相应的错误码。第 92 行对同步对象进行置 0 初始化。第 93 行对同步对象的任务位图进行初始化。第 94～100 行初始化同步对象名称。第 101 行记录同步对象是属于哪一个容器的。第 102 行设置 magic_number_变量以标识对象的有效性。第 103 行将分配出来的对象放入使用链表中。最后，第 105 行将对象通过输入参数_p_handle 返回给函数调用者。

3. 释放同步对象

当一个信号量或互斥锁不再需要时，需要释放所对应的同步对象。同步对象释放函数的实现如图 21.5 所示。

```
00109: error_t sync_object_free (sync_object_handle_t _handle)
00110: {
00111:     interrupt_level_t level;
00112:     sync_container_handle_t container;
00113:
00114:     if (is_in_interrupt () && STATE_UP == system_state ()) {
00115:         return ERROR_T (ERROR_SYNC_FREE_INVCONTEXT);
00116:     }
00117:     level = global_interrupt_disable ();
00118:     if (is_invalid_handle (_handle)) {
00119:         global_interrupt_enable (level);
00120:         return ERROR_T (ERROR_SYNC_FREE_INVHANDLE);
00121:     }
00122:     container = _handle->container_;
00123:     if (task_bitmap_is_empty (&_handle->pending_bitmap_)) {
00124:         // no task is pending on this object
00125:         _handle->magic_number_ = 0;
00126:         dll_remove (&container->used_, &_handle->node_);
00127:         dll_push_tail (&container->free_, &_handle->node_);
00128:         global_interrupt_enable (level);
00129:         return 0;
00130:     }
```

```
00131:     // there is/are task(s) pending on this object, wait it up one by one
00132:     do {
00133:         bit_t bit = task_bitmap_lowest_bit_get (&_handle->pending_bitmap_);
00134:         task_handle_t p_task = task_from_priority ((task_priority_t)bit);
00135:         task_bitmap_bit_clear (&_handle->pending_bitmap_, bit);
00136:         if (is_invalid_task (p_task)) {
00137:             continue;
00138:         }
00139:         p_task->ecode_ = ERROR_T (ERROR_SYNC_FREE_DELETED);
00140:         (void) task_state_change (p_task, TASK_STATE_READY);
00141:     } while (!task_bitmap_is_empty (&_handle->pending_bitmap_));
00142:     _handle->magic_number_ = 0;
00143:     dll_remove (&container->used_, &_handle->node_);
00144:     dll_push_tail (&container->free_, &_handle->node_);
00145:     global_interrupt_enable (level);
00146:     task_schedule (null);
00147:     return 0;
00148: }
```

图 21.5

第 118～121 行检查所需释放的同步对象句柄的有效性。当所需释放的同步对象上并不存在等待任务时，第 123～130 行的代码会被执行。其处理逻辑很简单，除了设置 magic_number_ 变量为 0 表示该对象不再有效外，还需要将同步对象从使用链表中移除并放入空闲链表中。

程序如果运行到了第 132 行，则说明被释放的同步对象上存在等待任务。在这种情形下，必须使所有等待任务都退出等待状态后才能删除同步对象。第 132～141 行遍历任务位图并让所有等待任务进入“就绪”状态（第 128 行）。第 133 行获取任务位图中优先级最高的任务。第 134 行通过优先级获得对应的任务。第 135 行从任务位图中清除当前正处理任务的优先级位。第 136 行检查任务是否已结束其生命周期，对于已删除的任务并不需要将其状态设置为“就绪”。第 139 行通过设置任务句柄中的 ecode_ 变量将错误码 ERROR_SYNC_FREE_DELETED 返回给等待任务。

第 142～144 行的功能与第 125～127 行是一样的。由于可能存在更高优先级的任务因为删除这个同步对象而进入就绪状态，所以需要在第 146 行调用 task_schedule()函数触发一次任务调度。

4．进入同步点

对于信号量，semaphore_take()函数的调用就意味着进入了同步对象的同步点。对于互斥锁，进入同步点是通过调用 mutex_lock()函数做到的。这两个函数最终都会调用 sync_point_enter()函数，其实现如图 21.6 所示。

```
00175: error_t sync_point_enter (sync_object_handle_t _handle, msecond_t _timeout)
00176: {
00177:     interrupt_level_t level;
00178:     sync_container_handle_t container;
00179:     task_handle_t p_task = task_self ();
00180:
00181:     if (is_in_interrupt ()) {
```

```
00182:         return ERROR_T (ERROR_SYNC_ENTER_INVCONTEXT);
00183:     }
00184:
00185:     level = global_interrupt_disable ();
00186:     if (is_invalid_handle (_handle)) {
00187:         global_interrupt_enable (level);
00188:         return ERROR_T (ERROR_SYNC_ENTER_INVHANDLE);
00189:     }
00190:     if (is_invalid_task (p_task)) {
00191:         global_interrupt_enable (level);
00192:         return ERROR_T (ERROR_SYNC_ENTER_INVTASK);
00193:     }
00194:     container = _handle->container_;
00195:     if (container->opt_.enter_ (_handle)) {
00196:         global_interrupt_enable (level);
00197:         return 0;
00198:     }
00199:     // the object isn't available and need to block the task
00200:     task_bitmap_bit_set (&_handle->pending_bitmap_, p_task->priority_);
00201:     p_task->timeout_ = _timeout;
00202:     (void) task_state_change (p_task, TASK_STATE_WAITING);
00203:     p_task->ecode_ = 0;
00204:     container->opt_.wait_ (_handle);
00205:     global_interrupt_enable (level);
00206:     task_schedule (null);
00207:     //lint -e{650}
00208:     if (ERROR_TASK_WAIT_TIMEOUT == MODULE_ERROR (p_task->ecode_)) {
00209:         p_task->ecode_ = ERROR_T (ERROR_SYNC_ENTER_TIMEOUT);
00210:     }
00211:     return p_task->ecode_;
00212: }
```

图 21.6

进入同步点必须是在任务环境中而不能在中断状态下，第 181～193 行代码正是预防这一点的。第 194 行获取同步对象所属容器，因为回调函数是保存在容器数据结构中的。

第 195 行调用进入同步点的回调函数并根据返回值判断是否继续后续操作。如果回调函数返回 true，则表示已经成功进入同步点，否则调用该函数的任务需进入等待状态。第 200 行将当前需要等待的任务优先级放入同步对象的任务位图中。第 201 行设置超时等待时间，这一时间在调用 task_state_change()函数将任务状态变为“等待”时有用。第 203 行将任务的返回错误码设置为 0，当任务被唤醒时需要通过检查 ecode_变量看是否存在等待超时。第 204 行调用等待回调函数。第 206 行触发的任务调度使得调用该函数的任务被暂停。

第 208 行的继续运行意味着任务从“等待”状态回到了“运行”状态。结束等待可能是因为所需等待的资源可用了，或者所指定的等待超时到期了，此时需要检查任务句柄中的 ecode_变量值来判断究竟是何种原因[②]。如果是因为等待超时而退出“等待”状态，那么在第 209 行将错误码重新设置为与同步对象相关的一个值。

② 当任务是因为等待超期而运行时，图 20.40 中的第 102 行将设置 ecode_变量的值。

除了 sync_point_enter()函数外，同步对象管理模块还提供了另一个函数——sync_point_try_to_enter()。这个函数只是尝试进入同步点，如果资源不可用，并不让调用任务进入“等待”状态，而是直接返回 ERROR_SYNC_ENTER_TRYAGAIN 错误码。该函数的实现其实就是 sync_point_enter()函数的一个片段，请读者自行查看源代码。

5．退出同步点

离开同步点时需要调用 sync_point_exit()函数，该函数被 semaphore_give()和 mutex_unlock()函数间接调用，其实现如图 21.7 所示。

```
00214: error_t sync_point_exit (sync_object_handle_t _handle)
00215: {
00216:     interrupt_level_t level;
00217:     sync_container_handle_t container;
00218:     error_t ecode = 0;
00219:
00220:     level = global_interrupt_disable ();
00221:     if (is_invalid_handle (_handle)) {
00222:         global_interrupt_enable (level);
00223:         return ERROR_T (ERROR_SYNC_LEAVE_INVHANDLE);
00224:     }
00225:     container = _handle->container_;
00226:     if (container->opt_.exit_ (_handle, &ecode)) {
00227:         global_interrupt_enable (level);
00228:         return 0;
00229:     }
00230:     if (0 != ecode) {
00231:         global_interrupt_enable (level);
00232:         return ecode;
00233:     }
00234:     // there is/are task(s) pending on this object, get the highest
00235:     // priority task to be READY
00236:     container->opt_.wake_ (_handle);
00237:     global_interrupt_enable (level);
00238:     task_schedule (null);
00239:     return 0;
00240: }
```

图 21.7

第 226 行调用离开回调函数并检查其返回值，如果返回值为 true，则说明没有其他的任务在等待该同步对象，可以直接返回；否则还需运行第 230 行开始的代码以唤醒处于“等待”状态的任务中优先级最高的那个。

第 230 行检查在离开同步点时是否存在错误，如果出现了错误则立即返回。第 236 行唤醒正在等待进入同步点的任务。在第 238 行触发一次任务调度，以便让可能被唤醒的更高优先级的任务获得运行机会。

21.1.2.2　信号量的实现

在同步对象实现的基础上，信号量的实现就相对简单了。其主要工作是初始化信号量容器，

实现信号量的几个回调函数并提供相应的信号量操作接口函数。图 21.8 示例说明了信号量的数据结构定义。

```
00032: typedef struct {
00033:     sync_object_t object_;
00034:     usize_t count_;
00035: } semaphore_t, *semaphore_handle_t;
```

图 21.8

注意：同步对象类型的变量 object_必须放在结构的头部。这样做使得我们可以直接通过强制类型转换将 semaphore_t 的地址转换成对应的同步对象地址。第 37 行定义了用于计数的变量 count_。

图 21.9 是信号量模块的全局变量定义。第 34 行定义了一个数组，这个数组中的每一个元素对应的就是一个信号量实例，也可以理解为由同步对象管理模块管理的同步对象，这些同步对象将被放入第 35 行定义的 g_semaphore_container 容器中。第 32 行的宏意味着最多支持 32 个信号量。

```
00032: #define CONFIG_MAX_SEMAPHORE   32
00033:
00034: static semaphore_t g_semaphore_pool [CONFIG_MAX_SEMAPHORE];
00035: static sync_container_t g_semaphore_container;
```

图 21.9

1．操作函数

信号量的创建、删除、获取、释放和检查计数等函数的实现可以从图 21.10 中找到。其中删除、获取和释放函数的实现非常简单，只需调用同步对象的对应函数即可。只有创建函数的实现需要讲解一下。

```
00105: static void semaphore_init ()
00106: {
00107:     sync_operation_t opt = {semaphore_callback_take, semaphore_callback_wait,
00108:         semaphore_callback_give, semaphore_callback_wake};
00109:     //lint -e{545}
00110:     (void) sync_container_init (&g_semaphore_container,
00111:         &g_semaphore_pool,CONFIG_MAX_SEMAPHORE, sizeof (semaphore_t), &opt);
00112: }
00113:
00114: error_t semaphore_create (semaphore_handle_t *_p_handle, const
00115:     char _name [],usize_t _count)
00116: {
00117:     static bool initialized = false;
00118:     interrupt_level_t level;
00119:     error_t ecode;
00120:
00121:     level = global_interrupt_disable ();
00122:     if (!initialized) {
```

```
00123:         semaphore_init ();
00124:         initialized = true;
00125:     }
00126:     global_interrupt_enable (level);
00127:     ecode = sync_object_alloc (&g_semaphore_container,
00128:     (sync_object_handle_t *)_p_handle, _name);
00129:     if (0 == ecode) {
00130:         (*_p_handle)->count_ = _count;
00131:     }
00132:     return ecode;
00133: }
00134:
00135: error_t semaphore_delete (semaphore_handle_t _handle)
00136: {
00137:     return sync_object_free (&_handle->object_);
00138: }
00139:
00140: error_t semaphore_try_to_take (semaphore_handle_t _handle)
00141: {
00142:     return sync_point_try_to_enter (&_handle->object_);
00143: }
00144:
00145: error_t semaphore_take (semaphore_handle_t _handle, msecond_t _timeout)
00146: {
00147:     return sync_point_enter (&_handle->object_, _timeout);
00148: }
00149:
00150: error_t semaphore_give (semaphore_handle_t _handle)
00151: {
00152:     return sync_point_exit (&_handle->object_);
00153: }
00154:
00155: //lint -e{818}
00156: usize_t semaphore_count_get (const semaphore_handle_t _handle)
00157: {
00158:     interrupt_level_t level;
00159:     usize_t count;
00160:
00161:     level = global_interrupt_disable ();
00162:     count = _handle->count_;
00163:     global_interrupt_enable (level);
00164:     return count;
00165: }
```

图 21.10

第 114～133 行实现了 semaphore_create()函数。该函数的主要工作除需要分配同步对象外，还有其他两项。一是注册信号量所实现的四个回调函数，即图中第 105～112 行 semaphore_init()函数中的实现；二是对信号量的计数变量初始化，即图中第 130 行的实现。

2. 同步对象回调函数

信号量的四个回调函数的实现如图 21.11 所示。

```
00037: static bool semaphore_callback_take (sync_object_handle_t _handle)
00038: {
00039:     semaphore_handle_t p_semaphore = (semaphore_handle_t) _handle;
```

```
00040:     if (0 != p_semaphore->count_) {
00041:         // semaphore is available, grab it
00042:         p_semaphore->count_ --;
00043:         return true;
00044:     }
00045:     return false;
00046: }
00047:
00048: static void semaphore_callback_wait (sync_object_handle_t _handle)
00049: {
00050:     UNUSED (_handle);
00051: }
00052:
00053: static bool semaphore_callback_give (sync_object_handle_t _handle,
00054:     error_t *_p_ecode)
00055: {
00056:     semaphore_handle_t p_semaphore = (semaphore_handle_t) _handle;
00057:
00058:     UNUSED (_p_ecode);
00059:
00060:     if (task_bitmap_is_empty (&p_semaphore->object_.pending_bitmap_)) {
00061:         // no task is pending on this semaphore
00062:         p_semaphore->count_ ++;
00063:         return true;
00064:     }
00065:     return false;
00066: }
00067:
00068: static void semaphore_callback_wake (sync_object_handle_t _handle)
00069: {
00070:     task_handle_t p_task;
00071:     bit_t bit;
00072:
00073:     UNUSED (_handle);
00074:
00075:     bit = task_bitmap_lowest_bit_get (&_handle->pending_bitmap_);
00076:     task_bitmap_bit_clear (&_handle->pending_bitmap_, bit);
00077:     p_task = task_from_priority ((task_priority_t)bit);
00078:     (void) task_state_change (p_task, TASK_STATE_READY);
00079: }
```

图 21.11

第 37～46 行定义了 semaphore_callback_take()函数，其在 semaphore_take()函数被调用时间接调用。在第 40～45 行首先检查信号量的计数是否大于 0，如果大于 0，将计数减一并返回 true，表示成功获得了信号量；否则返回 false。前面指出，一旦返回 false 就需要让任务进入“等待”状态。

对于信号量，当一个任务需要进入“等待”状态时并不需要进行其他的额外操作，其实现全涵盖在同步对象中，所以等待回调函数 semaphore_callback_wait()的实现为空。

第 55～66 行实现了 semaphore_callback_give()函数，该函数通过 semaphore_give()函数间接调用。第 60～65 行先检查是否有其他任务正等待该信号量，如果没有则只需对计数变量进行加一并返回 true（第 63 行）；否则返回 false，表示离开同步点时需要唤醒正在等待的、优先级

最高的那个任务（第 65 行）。

当离开同步点时，如果有其他任务在等待操作该信号量就需要调用唤醒任务的回调函数 semaphore_callback_wake()。第 75～78 行是从等待任务位图中获取优先级最高的那个任务，并将该任务的状态设置为“就绪”。

3. 模块管理

信号量模块的管理主要体现在两个函数上：通过实现 semaphore_dump()函数帮助查看系统中所有信号量的状态，以及实现 module_semaphore()函数，以便在系统退出时检查是否存在信号量泄漏问题。semaphore_dump()函数的实现如图 21.12 所示。

```
embedded/code/platform/sync/[illegible]
00167: static bool semaphore_dump_for_each (dll_t *_p_dll, dll_node_t *_p_node,
00168:     void *_p_arg)
00169: {
00170:     //lint -e{740, 826}
00171:     semaphore_handle_t handle = (semaphore_handle_t)_p_node;
00172:
00173:     UNUSED (_p_dll);
00174:     UNUSED (_p_arg);
00175:
00176:     console_print (" Name: %s\n", handle->object_.name_);
00177:     console_print (" Count: %u\n", handle->count_);
00178:     console_print ("\n");
00179:     return true;
00180: }
00181:
00182: void semaphore_dump ()
00183: {
00184:     if (is_in_interrupt ()) {
00185:         return;
00186:     }
00187:
00188:     scheduler_lock ();
00189:     console_print ("\n\n");
00190:     console_print ("Summary\n");
00191:     console_print ("-------\n");
00192:     console_print (" Supported: %u\n", CONFIG_MAX_SEMAPHORE);
00193:     console_print (" Allocated: %u\n", dll_size (&g_semaphore_container.used_));
00194:     console_print (" .BSS Used: %u\n", ((address_t)&g_semaphore_container
00195:         - (address_t)g_semaphore_pool) + sizeof (g_semaphore_container));
00196:     console_print ("\n");
00197:     console_print ("Statistics\n");
00198:     console_print ("----------\n");
00199:     console_print (" No Object: %u\n", g_semaphore_container.stats_noobj_);
00200:     console_print ("\n");
00201:     console_print ("Semaphore Details\n");
00202:     console_print ("-----------------\n");
00203:     (void) dll_traverse (&g_semaphore_container.used_, semaphore_dump_for_each, 0);
00204:     console_print ("\n");
00205:     scheduler_unlock ();
00206: }
```

图 21.12

通过使用 semaphore_dump()函数，可以了解整个信号量模块大约使用了多少内存，以及每一个在用信号量的状态。对其实现在此不做更多的解释。图 21.13 示例说明了 module_semaphore()函数的实现。

```
00081: static bool semaphore_check_for_each (dll_t *_p_dll, dll_node_t *_p_node,
           void *_p_arg)
00082: {
00083:     //lint -e{740, 826}
00084:     semaphore_handle_t handle = (semaphore_handle_t)_p_node;
00085:
00086:     UNUSED (_p_dll);
00087:     UNUSED (_p_arg);
00088:
00089:     console_print ("Error: semaphore \"%s\" isn't deleted\n",
00090:         handle->object_.name_);
00091:     return true;
00092: }
00093:
00094: error_t module_semaphore (system_state_t _state)
00095: {
00096:     if (STATE_DESTROYING == _state) {
00097:         // check whether all semaphores created have been deleted or
00098:         // not, if not take them as error
00099:         (void) dll_traverse (&g_semaphore_container.used_,
00100:             semaphore_check_for_each, 0);
00101:     }
00102:     return 0;
00103: }
```

图 21.13

module_semaphore()函数在系统退出时会以回调的形式被调用。在终止化时，信号量模块将检查是否所有的信号量都已释放。第 81～92 行所定义的 semaphore_check_for_each()函数用于遍历没有释放的信号量，并以错误日志的形式加以提示。

21.1.3　semaphore 示例程序

semaphore 示例程序的源代码如图 21.14 所示，其中创建了一个生产者和一个消费者共两个任务。生产者任务通过调用 semaphore_give()函数给消费者任务发信号，消费者任务则调用 semaphore_take()函数永久等待信号。

```
00026: #include "main.h"
00027: #include "device.h"
00028: #include "semaphore.h"
00029: #include "console.h"
00030:
00031: static void task_consumer (const char _name [], void *_p_arg)
00032: {
00033:     semaphore_handle_t semaphore = (semaphore_handle_t)_p_arg;
00034:
00035:     console_print ("%s: going to take semaphore\n", _name);
```

```
00036:     (void) semaphore_take (semaphore, WAIT_FOREVER);
00037:     console_print ("%s: taken\n", _name);
00038: }
00039:
00040: static void task_producer (const char _name [], void *_p_arg)
00041: {
00042:     semaphore_handle_t semaphore = (semaphore_handle_t)_p_arg;
00043:
00044:     console_print ("%s: going to give semaphore\n", _name);
00045:     (void) semaphore_give (semaphore);
00046:     console_print ("%s: given\n", _name);
00047:     (void) task_sleep (1000);
00048:
00049:     semaphore_dump ();
00050:     multitasking_stop ();
00051: }
00052:
00053: error_t module_testapp (system_state_t _state)
00054: {
00055:     static task_handle_t producer;
00056:     static task_handle_t consumer;
00057:     STACK_DECLARE (stack_for_producer, 1024);
00058:     STACK_DECLARE (stack_for_consumer, 1024);
00059:     static semaphore_handle_t semaphore;
00060:     static device_handle_t ctrlc_handle;
00061:
00062:     if (STATE_INITIALIZING == _state) {
00063:         (void) semaphore_create (&semaphore, "Test", 0);
00064:
00065:         (void) task_create (&producer, "Producer", 16,
00066:             stack_for_producer, sizeof (stack_for_producer));
00067:         (void) task_start (producer, task_producer, semaphore);
00068:
00069:         (void) task_create (&consumer, "Consumer", 11,
00070:             stack_for_consumer, sizeof (stack_for_consumer));
00071:         (void) task_start (consumer, task_consumer, semaphore);
00072:
00073:         (void) device_open (&ctrlc_handle, "/dev/ui/ctrlc", 0);
00074:     }
00075:     else if (STATE_DESTROYING == _state) {
00076:         (void) device_close (ctrlc_handle);
00077:         (void) task_delete (consumer);
00078:         (void) task_delete (producer);
00079:         (void) semaphore_delete (semaphore);
00080:     }
00081:     return 0;
00082: }
00083:
00084: int module_registration_entry (int argc, char *argv [])
00085: {
00086:     UNUSED (argc);
00087:     UNUSED (argv);
00088:
00089:     (void) module_register ("Interrupt", MODULE_INTERRUPT, CPU_LEVEL,
            module_interrupt);
00090:     (void) module_register ("Device", MODULE_DEVICE, DRIVER_LEVEL, module_device);
00091:     (void) module_register ("Timer", MODULE_TIMER, OS_LEVEL, module_timer);
00092:     (void) module_register ("Task", MODULE_TASK, OS_LEVEL, module_task);
00093:     (void) module_register ("Semaphore", MODULE_SEMAPHORE, OS_LEVEL,
            module_semaphore);
```

```
00094:     (void) module_register ("TestApp", MODULE_TESTAPP, APPLICATION_LEVEL3,
               module_testapp);
00095:     return 0;
00096: }
```

图 21.14

示例程序的运行结果列于图 21.15 中。从图中可以看出，由于消费者任务的优先级更高，所以它先获得运行机会。因为信号量的开始计数为 0，所以它进入“等待”状态，而这造成生产者任务获得运行机会。当生产者任务调用 semaphore_give()函数给信号时，由于它的优先级比消费者低，这会造成消费者任务立即“抢占”到处理器而运行。消费者任务在获得信号量后，继续运行在终端上打印出“taken”，并通过让 task_consumer()函数返回的方式结束任务，这使得生产者任务得以再次运行。生产者任务在终端上打印“given”，并调用 semaphore_dump()函数显示信号量模块的状态，以及最后调用 multitasking_stop()函数结束多任务环境。

```
                          /embedded/build
$ make
                          /embedded/build
$ ./release/semaphore.exe

System is going to be up

Info: press Ctrl+C to terminate!
Consumer: going to take semaphore
Producer: going to give semaphore
Consumer: taken
Producer: given

Summary
-------
  Supported: 32
  Allocated: 1
  .BSS Used: 1840

Statistics
----------
  No Object: 0

Semaphore Details
-----------------
  Name: Test
    Count: 0

System is going to be down
```

图 21.15

21.2 互斥锁

与信号量作为资源“计数器”完全不同的是，互斥锁（mutex）体现的是“有你没我”的“不合作精神”。由于是“锁”，所以存在“上锁”和“解锁”两个不同的操作。

21.2.1　应用场合

如果程序中存在一个链表，且多个任务需要对链表进行存取，为了防止出现竞争问题，需要使用一定的方法来保证对链表的操作做到“原子性”。在这种情形下，可以通过互斥锁来实现目的。

采用互斥锁对共享资源进行保护具有以下几个特点。

- 当一个任务在对资源进行访问时，其他的任务只能等待它完成对资源的访问才有可能获得访问权。
- 对共享资源的访问权是成功调用 mutex_lock()函数完成上锁开始的。反之，成功调用 mutex_unlock()函数进行解锁标志着完成了对共享资源的访问，也是其他任务获得访问权的时机。
- 任务对互斥锁上锁后，它必须“亲自”对其进行解锁，而不能将解锁动作交于其他任务“代办”。
- 互斥锁不能运用于中断状态，因此它不能用做任务与中断间的通信手段。

21.2.2　程序实现

互斥锁的实现是基于前面介绍的同步对象模块的。图 21.16 是互斥锁程序实现的头文件。

```
00033: #ifndef __task_handle_defined__
00034: struct type_task;
00035: typedef struct type_task task_t, *task_handle_t;
00036: #define __task_handle_defined__
00037: #endif
00038:
00039: typedef struct {
00040:     sync_object_t object_;
00041:     task_handle_t owner_;
00042: } mutex_t, *mutex_handle_t;
```

图 21.16

第 39～42 行定义了互斥锁的管理数据结构。同样地，第 40 行的 object_变量需要像信号量实现那样放在数据结构的最开始处。当互斥锁处于锁定状态时，第 41 行定义的任务句柄变量用于记录锁的“持有者”，这一信息在解锁时用于防止“代办”。图 21.17 定义了互斥锁的实例及容器。

```
00031: #define CONFIG_MAX_MUTEX     32
00032:
00033: static mutex_t g_mutex_pool [CONFIG_MAX_MUTEX];
00034: static sync_container_t g_mutex_container;
```

图 21.17

21.2.2.1 操作函数

互斥锁的操作函数列于图 21.18 中，它的实现与上一节的信号量除了名字存在差异外，其余的几乎相同，因此在这里不再一一讲解。

```
00111: static void mutex_init ()
00112: {
00113:     sync_operation_t opt = {mutex_callback_lock,
00114:         mutex_callback_wait, mutex_callback_unlock, mutex_callback_wake};
00115:     //lint -e{545}
00116:     (void) sync_container_init (&g_mutex_container, &g_mutex_pool,
00117:         CONFIG_MAX_MUTEX, sizeof (mutex_t), &opt);
00118: }
00119:
00120: error_t mutex_create (mutex_handle_t *_p_handle, const char _name [])
00121: {
00122:     static bool initialized = false;
00123:     interrupt_level_t level;
00124:     error_t ecode;
00125:
00126:     level = global_interrupt_disable ();
00127:     if (!initialized) {
00128:         mutex_init ();
00129:         initialized = true;
00130:     }
00131:     global_interrupt_enable (level);
00132:
00133:     ecode = sync_object_alloc (&g_mutex_container,
00134:         (sync_object_handle_t *) _p_handle, _name);
00135:     if (0 == ecode) {
00136:         (*_p_handle)->owner_ = null;
00137:     }
00138:     return ecode;
00139: }
00140:
00141: error_t mutex_delete (mutex_handle_t _handle)
00142: {
00143:     return sync_object_free (&_handle->object_);
00144: }
00145:
00146: error_t mutex_try_to_lock (mutex_handle_t _handle)
00147: {
00148:     return sync_point_try_to_enter (&_handle->object_);
00149: }
00150:
00151: error_t mutex_lock (mutex_handle_t _handle, msecond_t _timeout)
00152: {
00153:     return sync_point_enter (&_handle->object_, _timeout);
00154: }
00155:
00156: error_t mutex_unlock (mutex_handle_t _handle)
00157: {
00158:     return sync_point_exit (&_handle->object_);
00159: }
```

图 21.18

21.2.2.2　同步对象回调函数

互斥锁与信号量之间的不同完全反映在四个同步对象回调函数的实现上，如图 21.19 所示。

```
embedded/code/platform/sync/[illegible]
00036: static bool mutex_callback_lock (sync_object_handle_t _handle)
00037: {
00038:     mutex_handle_t p_mutex = (mutex_handle_t) _handle;
00039:
00040:     if (null == p_mutex->owner_) {
00041:         // grab the mutex
00042:         p_mutex->owner_ = task_self ();
00043:         return true;
00044:     }
00045:     return false;
00046: }
00047:
00048: static void mutex_callback_wait (sync_object_handle_t _handle)
00049: {
00050:     UNUSED (_handle);
00051: }
00052:
00053: static bool mutex_callback_unlock (sync_object_handle_t _handle,
00054:     error_t *_p_ecode)
00055: {
00056:     mutex_handle_t p_mutex = (mutex_handle_t) _handle;
00057:     task_handle_t p_task = task_self ();
00058:
00059:     // grab the mutex
00060:     if (is_in_interrupt () || is_invalid_task (p_task)) {
00061:         return ERROR_T (ERROR_MUTEX_INVCONTEXT);
00062:     }
00063:     if (p_mutex->owner_ != p_task) {
00064:         *_p_ecode = ERROR_T (ERROR_MUTEX_NOTOWNER);
00065:         return false;
00066:     }
00067:
00068:     if (task_bitmap_is_empty (&p_mutex->object_.pending_bitmap_)) {
00069:         // no task is pending on this mutex
00070:         p_mutex->owner_ = null;
00071:         return true;
00072:     }
00073:     return false;
00074: }
00075:
00076: static void mutex_callback_wake (sync_object_handle_t _handle)
00077: {
00078:     mutex_handle_t p_mutex = (mutex_handle_t) _handle;
00079:     task_handle_t p_task;
00080:     bit_t bit;
00081:
00082:     bit = task_bitmap_lowest_bit_get (&_handle->pending_bitmap_);
00083:     task_bitmap_bit_clear (&_handle->pending_bitmap_, bit);
00084:     p_task = task_from_priority ((task_priority_t)bit);
00085:     (void) task_state_change (p_task, TASK_STATE_READY);
00086:     p_mutex->owner_ = p_task;
00087: }
```

图 21.19

当调用 mutex_lock()函数对互斥锁进行上锁操作时，回调函数 mutex_callback_lock()会被间接调用，它实现于第 36～46 行。第 40～45 行首先检查锁是否被某任务所持有，如果没有，则说明这个锁可以立即被调用任务获得。在第 42 行记录调用任务的句柄后返回 true，表示调用任务已成功获得锁。如果锁已经被某任务所持有则返回 false，这将导致 mutex_callback_wait()回调函数会被调用以使任务进入“等待”状态。其实现位于第 48～51 行，目前为空。

第 53～74 行是 mutex_callback_unlock()函数的实现，在对互斥锁进行解锁操作时，这个回调函数将被间接调用。第 60～62 行防止 mutex_lock()函数在中断状态被调用，如果不是则返回错误。第 63～66 行检查进行解锁操作的任务是否是锁的持有者。第 68～73 行检查所释放的锁上是否有其他任务在等待，如果没有则直接返回 true，表示解锁操作完成了；否则返回 false，表示需要唤醒正在等待的、优先级最高的任务。

如果在解完锁后有其他的任务正等待锁，则 mutex_callback_wake()函数将被调用。其实现与信号量中的很相似，且在最后第 86 行需要将被唤醒的任务设置为互斥锁的持有者。

21.2.2.3 模块管理

通过使用 mutex_dump()函数可以查看系统中所有互斥锁的状态，以及整个互斥锁模块管理数据所使用的大致内存数量和统计信息，其实现如图 21.20 所示。

```
00162: static bool mutex_dump_for_each (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00163: {
00164:     //lint -e{740, 826}
00165:     mutex_handle_t handle = (mutex_handle_t)_p_node;
00166:
00167:     UNUSED (_p_dll);
00168:     UNUSED (_p_arg);
00169:
00170:     console_print (" Name: %s\n", handle->object_.name_);
00171:     if (0 == handle->owner_) {
00172:         console_print (" Owner: null\n");
00173:     }
00174:     else {
00175:         console_print (" Owner: %s\n", handle->owner_->name_);
00176:     }
00177:     console_print ("\n");
00178:     return true;
00179: }
00180:
00181: void mutex_dump ()
00182: {
00183:     if (is_in_interrupt ()) {
00184:         return;
00185:     }
00186:
00187:     scheduler_lock ();
00188:     console_print ("\n\n");
00189:     console_print ("Summary\n");
00190:     console_print ("-------\n");
00191:     console_print (" Supported: %u\n", CONFIG_MAX_MUTEX);
```

```
00192:     console_print (" Allocated: %u\n", dll_size (&g_mutex_container.used_));
00193:     console_print (" .BSS Used: %u\n", ((address_t)&g_mutex_container
00194:         - (address_t)g_mutex_pool) + sizeof (g_mutex_container));
00195:     console_print ("\n");
00196:     console_print ("Statistics\n");
00197:     console_print ("----------\n");
00198:     console_print (" No Object: %u\n", g_mutex_container.stats_noobj_);
00199:     console_print ("\n");
00200:     console_print ("Mutex Details\n");
00201:     console_print ("-------------\n");
00202:     (void) dll_traverse (&g_mutex_container.used_, mutex_dump_for_each, 0);
00203:     console_print ("\n");
00204:     scheduler_unlock ();
00205: }
```

图 21.20

与信号量相类似，互斥锁模块也只关心系统的终止化过程，其模块回调函数实现如图 21.21 所示。通过在终止化过程中检查哪些互斥锁还没有被释放的方式，帮助我们查找潜在的资源泄漏问题。

```
00089: static bool mutex_check_for_each (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00090: {
00091:     //lint -e{740, 826}
00092:     mutex_handle_t handle = (mutex_handle_t)_p_node;
00093:
00094:     UNUSED (_p_dll);
00095:     UNUSED (_p_arg);
00096:
00097:     console_print ("Error: mutex \"%s\" isn't deleted\n", handle->object_.name_);
00098:     return true;
00099: }
00100:
00101: error_t module_mutex (system_state_t _state)
00102: {
00103:     if (STATE_DESTROYING == _state) {
00104:         // check whether all mutics created have been deleted or not,
00105:         // if not take them as error
00106:         (void) dll_traverse (&g_mutex_container.used_, mutex_check_for_each, 0);
00107:     }
00108:     return 0;
00109: }
```

图 21.21

21.2.3　mutex 示例程序

mutex 示例程序的源代码和运行结果分别列于图 21.22 和图 21.23 中，请读者自行对照运行结果阅读代码。

```
00026: #include "main.h"
00027: #include "device.h"
00028: #include "mutex.h"
00029: #include "console.h"
```

```
00030:
00031: static void task_high (const char _name [], void *_p_arg)
00032: {
00033:    mutex_handle_t mutex = (mutex_handle_t)_p_arg;
00034:
00035:    console_print ("%s: going to take lock\n", _name);
00036:    (void) mutex_lock (mutex, WAIT_FOREVER);
00037:    console_print ("%s: lock is taken\n", _name);
00038:    console_print ("%s: going to sleep\n", _name);
00039:    (void) task_sleep (1000);
00040:    console_print ("%s: is waken\n", _name);
00041:    console_print ("%s: going to free lock\n", _name);
00042:    (void) mutex_unlock (mutex);
00043:    console_print ("%s: lock is freed\n", _name);
00044: }
00045:
00046: static void task_low (const char _name [], void *_p_arg)
00047: {
00048:    mutex_handle_t mutex = (mutex_handle_t)_p_arg;
00049:
00050:    console_print ("%s: going to take lock\n", _name);
00051:    (void) mutex_lock (mutex, WAIT_FOREVER);
00052:    console_print ("%s: lock is taken\n", _name);
00053:    console_print ("%s: going to free lock\n", _name);
00054:    (void) mutex_unlock (mutex);
00055:    console_print ("%s: lock is freed\n", _name);
00056:
00057:    mutex_dump ();
00058:    multitasking_stop ();
00059: }
00060:
00062: error_t module_testapp (system_state_t _state)
00063: {
00064:    static task_handle_t high;
00065:    static task_handle_t low;
00066:    STACK_DECLARE (stack_for_high, 1024);
00067:    STACK_DECLARE (stack_for_low, 1024);
00068:    static mutex_handle_t mutex;
00069:    static device_handle_t ctrlc_handle;
00070:
00071:    if (STATE_INITIALIZING == _state) {
00072:       (void) mutex_create (&mutex, "Test");
00073:
00074:       (void) task_create (&low, "Task Low", 16, stack_for_low,
              sizeof (stack_for_low));
00075:       (void) task_start (low, task_low, mutex);
00076:
00077:       (void) task_create (&high, "Task High", 11, stack_for_high,
              sizeof (stack_for_high));
00078:       (void) task_start (high, task_high, mutex);
00079:
00080:       (void) device_open (&ctrlc_handle, "/dev/ui/ctrlc", 0);
00081:    }
00082:    else if (STATE_DESTROYING == _state) {
00083:       (void) device_close (ctrlc_handle);
00084:       (void) task_delete (high);
00085:       (void) task_delete (low);
00086:       (void) mutex_delete (mutex);
00087:    }
00088:    return 0;
```

```
00089: }
00090:
00091: int module_registration_entry (int argc, char *argv [])
00092: {
00093:     UNUSED (argc);
00094:     UNUSED (argv);
00095:
00096:     (void) module_register ("Interrupt", MODULE_INTERRUPT, CPU_LEVEL,
           module_interrupt);
00097:     (void) module_register ("Device", MODULE_DEVICE, DRIVER_LEVEL, module_device);
00098:     (void) module_register ("Timer", MODULE_TIMER, OS_LEVEL, module_timer);
00099:     (void) module_register ("Task", MODULE_TASK, OS_LEVEL, module_task);
00100:     (void) module_register ("Mutex", MODULE_MUTEX, OS_LEVEL, module_mutex);
00101:     (void) module_register ("TestApp", MODULE_TESTAPP, APPLICATION_LEVEL3,
           module_testapp);
00102:     return 0;
00103: }
```

图 21.22

```
                        /embedded/build
$ make
                        /embedded/build
$ ./release/mutex.exe

System is going to be up

Info: press Ctrl+C to terminate!
Task High: going to take lock
Task High: lock is taken
Task High: going to sleep
Task Low: going to take lock
Task High: is waken
Task High: going to free lock
Task High: lock is freed
Task Low: lock is taken
Task Low: going to free lock
Task Low: lock is freed

Summary
-------
  Supported: 32
  Allocated: 1
  .BSS Used: [illegible]

Statistics
----------
  No object: 0

Mutex Details
-------------
  Name: Test
    Owner: null

System is going to be down
```

图 21.23

21.2.4　优先级反转与继承

优先级反转这一概念是嵌入式软件开发工程师必须掌握的，这也有助于我们更好地理解实

时操作系统中的“抢占”行为。

21.2.4.1　优先级反转示例程序

探索优先级反转问题需要先通过 prioinv 示例程序，其源程序如图 21.24 所示。这个例子中用到了后面将要讲解的事件（参见 21.3 节），我们将结合运行结果来了解什么是优先级反转问题。

```
00026: #include "main.h"
00027: #include "device.h"
00028: #include "mutex.h"
00029: #include "event.h"
00030: #include "console.h"
00031:
00032: #define EVENT    0x01
00033:
00034: static task_handle_t g_task_high;
00035: static task_handle_t g_task_middle;
00036: static task_handle_t g_task_low;
00037:
00038: STACK_DECLARE (g_stack_for_high, 1024);
00039: STACK_DECLARE (g_stack_for_middle, 1024);
00040: STACK_DECLARE (g_stack_for_low, 1024);
00041:
00042: static void task_high (const char _name [], void *_p_arg)
00043: {
00044:     mutex_handle_t mutex = (mutex_handle_t) _p_arg;
00045:
00046:     console_print ("%s: yeild CPU\n", _name);
00047:     (void) task_sleep (0);
00048:     console_print ("%s: yeild CPU again\n", _name);
00049:     (void) task_sleep (0);
00050:     console_print ("%s: trying to lock\n", _name);
00051:     (void) mutex_lock (mutex, WAIT_FOREVER);
00052:     console_print ("%s: locked\n", _name);
00053:     (void) mutex_unlock (mutex);
00054:     console_print ("%s: unlocked\n", _name);
00055: }
00056:
00057: static void task_middle (const char _name [], void *_p_arg)
00058: {
00059:     event_set_t received;
00060:
00061:     UNUSED (_p_arg);
00062:
00063:     console_print ("%s: receiving event\n", _name);
00064:     (void) event_receive (EVENT, &received, WAIT_FOREVER,
00065:         EVENT_WAIT_ANY | EVENT_RETURN_EXPECTED);
00066:     console_print ("%s: event received\n", _name);
00067: }
00068:
00069: static void task_low (const char _name [], void *_p_arg)
00070: {
00071:     mutex_handle_t mutex = (mutex_handle_t) _p_arg;
00072:
00073:     console_print ("%s: trying to lock\n", _name);
00074:     (void) mutex_lock (mutex, WAIT_FOREVER);
00075:     console_print ("%s: locked\n", _name);
```

```
00076:     console_print ("%s: sending event\n", _name);
00077:     (void) event_send (g_task_middle, EVENT);
00078:     console_print ("%s: event sent\n", _name);
00079:     (void) mutex_unlock (mutex);
00080:     console_print ("%s: unlocked\n", _name);
00081:     multitasking_stop ();
00082: }
00083:
00084: error_t module_testapp (system_state_t _state)
00085: {
00086:     static mutex_handle_t mutex;
00087:     static device_handle_t ctrlc_handle;
00088:
00089:     if (STATE_INITIALIZING == _state) {
00090:         (void) mutex_create (&mutex, "Test");
00091:         (void) task_create (&g_task_low, "Low", 16, g_stack_for_low,
00092:             sizeof (g_stack_for_low));
00093:         (void) task_start (g_task_low, task_low, mutex);
00094:
00095:         (void) task_create (&g_task_middle, "Middle", 13,
00096:             g_stack_for_middle, sizeof (g_stack_for_middle));
00097:         (void) task_start (g_task_middle, task_middle, mutex);
00098:
00099:         (void) task_create (&g_task_high, "High", 11,
00100:             g_stack_for_high, sizeof (g_stack_for_high));
00101:         (void) task_start (g_task_high, task_high, mutex);
00102:
00103:         (void) device_open (&ctrlc_handle, "/dev/ui/ctrlc", 0);
00104:     }
00105:     else if (STATE_DESTROYING == _state) {
00106:         (void) device_close (ctrlc_handle);
00107:         (void) task_delete (g_task_high);
00108:         (void) task_delete (g_task_middle);
00109:         (void) task_delete (g_task_low);
00110:         (void) mutex_delete (mutex);
00111:     }
00112:     return 0;
00113: }
00114:
00115: int module_registration_entry (int argc, char *argv [])
00116: {
00117:     UNUSED (argc);
00118:     UNUSED (argv);
00119:
00120:     (void) module_register ("Interrupt", MODULE_INTERRUPT, CPU_LEVEL,
             module_interrupt);
00121:     (void) module_register ("Device", MODULE_DEVICE, DRIVER_LEVEL, module_device);
00122:     (void) module_register ("Timer", MODULE_TIMER, OS_LEVEL, module_timer);
00123:     (void) module_register ("Task", MODULE_TASK, OS_LEVEL, module_task);
00124:     (void) module_register ("Mutex", MODULE_MUTEX, OS_LEVEL, module_mutex);
00125:     (void) module_register ("TestApp", MODULE_TESTAPP, APPLICATION_LEVEL3,
             module_testapp);
00126:     return 0;
00127: }
```

图 21.24

prioinv 示例程序的运行结果如图 21.25 所示。为了方便解释优先级反转问题，需要借助时序图来示例说明其中三个不同优先级任务在不同时刻的运行状态，如图 21.26 所示。

```
                     /embedded/build
$ make
                     /embedded/build
$ ./release/prioinv.exe

System is going to be up

Info: press Ctrl+C to terminate!
High: yeild CPU
Middle: receiving event
High: yeild CPU again
Low: trying to lock
Low: locked
Low: sending event
High: trying to lock
Middle: event received
Low: event sent
High: locked
High: unlocked
Low: unlocked

System is going to be down
```

图 21.25

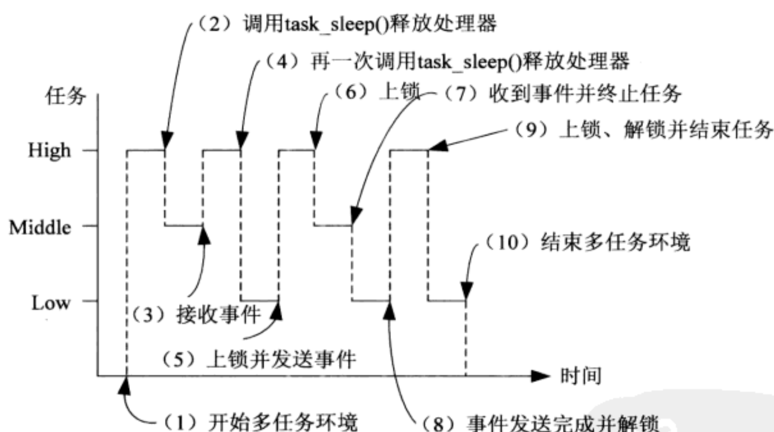


图 21.26

在 prioinv 示例程序中存在三个任务。示例程序的运行将按时间先后顺序发生以下几个步骤。

（1）系统进入多任务环境，高优先级的任务获得运行机会。

（2）在程序第 47 行，高优先级的任务调用 task_sleep()函数放弃一次运行机会。这导致发生一次任务切换使得中优先级的任务获得运行机会。

（3）在程序第 64 行，中优先级的任务调用 event_receive()函数接收所希望的事件，由于事件还没有发生，任务被暂停并进入“等待”状态。此刻又触发一次任务调度，调度的结果是高优先级的任务再一次获得处理器而运行。

（4）在程序第 49 行，高优先级的任务再一次调用 task_sleep()函数第二次放弃处理器。此刻触发的任务调度因为中优先级的任务还在等待事件，这使得低优先级的任务获得运行机会。

（5）低优先级的任务在第 74 行调用 mutex_lock()函数对源程序在第 90 行所创建的互斥锁进行上锁操作。由于这是第一次对该锁进行上锁操作，所以能成功地获得锁。在上锁成功后，该任务在源程序的第 77 行调用 event_send()函数向中优先级的任务发送事件，这又触发一次任务调度。尽管低优先级的任务对 event_send()函数的调用会造成中优先级的任务退出“等待”状态，但此时由于高优先级的任务已完成了放弃一次处理器的操作也进入了“就绪”状态，因此调度结果是高优先级的任务会获得运行机会。

（6）高优先级的任务在源程序的第 51 行调用 mutex_lock()函数试图对第 90 行所分配的互斥锁进行上锁。而该锁正被最低优先级的任务所持有，因此高优先级的任务被迫进入“等待”状态而放弃处理器，所触发的任务调度使得中优先级的任务获得运行机会。

（7）当中优先级的任务因为收到事件而从 event_receive()函数返回后，它打印信息后让任务体函数返回，结果是中优先级的任务将退出而触发一次任务调度。此次调度使得获得了锁的低优先级的任务得到运行机会。

（8）低优先级的任务从 event_send()函数返回后，在源程序的第 79 行释放所持有的锁。释放操作使得高优先级的任务满足了运行条件而被调度运行。

（9）高优先级的任务在完成上锁和解锁操作后，也像中优先级的任务那样退出。由于只有低优先级的任务处于就绪状态，所以低优先级的任务得以继续运行。

（10）低优先级的任务从 mutex_unlock()函数返回后，在第 81 行调用 multitasking_stop()函数终止整个系统。

这个示例程序到底想说明什么问题呢？从时序图可以看出，中优先级的任务比高优先级的任务更先结束运行。而理论上，高优先级的任务应当比中优先级的任务更早结束运行。这种因为高优先级的任务等待低优先级的任务而使得响应被延时的现象就是实时系统中“著名的”优先级反转问题。

在该示例程序中，对于高优先级的任务我们采用 task_sleep()的方式放弃处理器而制造出优先级反转问题，这是因为 ClearRTOS 并不能接管处理器的中断。而在真实环境的实时操作系统中，优先级反转问题会因为中断的发生而轻易地出现。另外，示例程序中只列出了三个任务，如果任务数量更多，高优先级的任务因为优先级反转而导致的延时将进一步恶化。

解决优先级反转的思路是，需要动态地改变任务的优先级。当高优先级的任务等待低优先级的任务释放正占用的资源时，需要将该低优先级任务的优先级提高到与高优先级任务一样。当低优先级任务的优先级变高后，它将在后续的任务调度中获得更好的响应，因而能尽早地使用完高优先级任务所等待的资源。一旦低优先级的任务释放了高优先级的任务所等待的资源

后，其优先级就恢复到原值。这种为了解决优先级反转问题而动态调整任务优先级的方法就是优先级继承。

21.2.4.2 实现优先级继承

为了实现优先级继承功能，需要在现有互斥锁的实现上做一定的改动。所有的改动示例说明于图 21.27 中。

```
embedded/code/platform/sync/v2/inc/mutex.h
00044: typedef struct {
00045:     sync_object_t object_;
00046:     task_handle_t owner_;
00047:     bool inherited_;
00048:     task_priority_t original_;
00050: } mutex_t, *mutex_handle_t;
00051:
```

```
embedded/code/platform/sync/v2/src/mutex.c
00036: static bool mutex_callback_lock (sync_object_handle_t _handle)
00037: {
00038:     mutex_handle_t p_mutex = (mutex_handle_t) _handle;
00039:
00040:     if (null == p_mutex->owner_) {
00041:         // grab the mutex
00042:         p_mutex->owner_ = task_self ();
00043:         p_mutex->inherited_ = false;
00044:         return true;
00045:     }
00046:     return false;
00047: }
00048:
00049: static void mutex_callback_wait (sync_object_handle_t _handle)
00050: {
00051:     mutex_handle_t p_mutex = (mutex_handle_t) _handle;
00052:     task_handle_t p_task = task_self ();
00053:
00054:     if (p_mutex->owner_->priority_ > p_task->priority_) {
00055:         if (!p_mutex->inherited_) {
00056:             p_mutex->original_ = p_mutex->owner_->priority_;
00057:             p_mutex->inherited_ = true;
00058:         }
00059:         task_priority_change (p_mutex->owner_, p_task->priority_);
00060:     }
00061: }
00062:
00063: static bool mutex_callback_unlock (sync_object_handle_t _handle,
00064:     error_t *_p_ecode)
00065: {
00066:     mutex_handle_t p_mutex = (mutex_handle_t) _handle;
00067:     task_handle_t p_task = task_self ();
00068:
00069:     // grab the mutex
00070:     if (is_in_interrupt () || is_invalid_task (p_task)) {
00071:         return ERROR_T (ERROR_MUTEX_INVCONTEXT);
00072:     }
00073:     if (p_mutex->owner_ != p_task) {
00074:         *_p_ecode = ERROR_T (ERROR_MUTEX_NOTOWNER);
```

```
00075:         return false;
00076:     }
00077:     // restore owner's original priority
00078:     if (p_mutex->inherited_) {
00079:         p_mutex->inherited_ = false;
00080:         task_priority_change (p_mutex->owner_, p_mutex->original_);
00081:     }
00082:     if (task_bitmap_is_empty (&p_mutex->object_.pending_bitmap_)) {
00083:         // no task is pending on this mutex
00084:         p_mutex->owner_ = null;
00085:         return true;
00086:     }
00087:     return false;
00088: }
00134: error_t mutex_create (mutex_handle_t *_p_handle, const char _name [])
00135: {
00136:     static bool initialized = false;
00137:     interrupt_level_t level;
00138:     error_t ecode;
00139:
00140:     level = global_interrupt_disable ();
00141:     if (!initialized) {
00142:         mutex_init ();
00143:         initialized = true;
00144:     }
00145:     global_interrupt_enable (level);
00146:         ecode = sync_object_alloc (&g_mutex_container,
00147:             (sync_object_handle_t *) _p_handle, _name);
00148:     if (0 == ecode) {
00149:         (*_p_handle)->owner_ = null;
00150:         (*_p_handle)->inherited_ = false;
00151:     }
00152:     return ecode;
00153: }
```

图 21.27

第一处改动是 mutex.h 中的第 47～48 行。增加的 inherited_变量用于表示互斥锁是否运用了优先级继承功能，original_变量用于记录任务的原始优先级。

其他的改动都位于 mutex.c 文件中。在第 43 行，当一个任务获得互斥锁时， inherited_变量设置为 false，表示没有运用优先级继承。mutex_callback_wait()函数的实现在之前是空的，现在增加了第 51～60 行的实现。它的被调用意味着互斥锁已经被某任务所持有了，因此第 51～60 行的作用就是检查是否需要进行优先级继承。如果需要，则在保存锁持有任务的原始优先级后，通过调用 task_priority_change()函数提高锁持有任务的优先级。

mutex_callback_unlock()函数中新增的第 78～81 行用于恢复任务的原始优先级，这里同样需要使用到 task_priority_change()函数。mutex_create()函数中新增了第 150 行，用于对 inherited_变量进行初始化。

21.2.4.3　任务优先级更新

任务优先级更新函数 task_priority_change()的实现如图 21.28 所示。

```
00588: void task_priority_change (task_handle_t _handle, task_priority_t _to)
00589: {
00590:     if ((TASK_STATE_READY == _handle->state_) ||
00591:         (TASK_STATE_RUNNING == _handle->state_)) {
00592:         task_bitmap_bit_clear (&g_ready_bitmap, _handle->priority_);
00593:         g_priority_map [_handle->priority_] = null;
00594:         _handle->priority_ = _to;
00595:         task_bitmap_bit_set (&g_ready_bitmap, _handle->priority_);
00596:         g_priority_map [_handle->priority_] = _handle;
00597:     }
00598:     else {
00599:         _handle->priority_ = _to;
00600:     }
00601: }
```

图 21.28

第 590～591 行首先判断任务是否处于“就绪”或“运行”状态，如果是，则在第 592～596 行需要对就绪任务位图 g_ready_bitmap 和优先级映射数组 g_priority_map 进行更新。其中 g_priority_map 数组就是为了实现优先级继承功能而引入的，否则这个数组完全不需要。如果任务不处于“就绪”或“运行”状态，则只要在第 599 行修改任务的优先级就行了。

21.2.4.4 优先级继承示例程序

对互斥锁模块实现优先级继承后，我们通过运行另一个示例程序来观察改动后的效果——prioinh。prioinh 示例程序的源代码与 prioinv 是完全相同的，只是在编译时链接的是 libsyncv2.a（实现了优先级继承）而不是之前的 libsyncv1.a（无优先级继承实现）。prioinh 示例程序的运行结果如图 21.29 所示。同样为了方便理解，在图 21.30 中通过时序图示例说明了每一个任务切换点。

```
                    /embedded/build
$ make
                    /embedded/build
$ ./release/prioinh.exe

System is going to be up

Info: press Ctrl+C to terminate!
High: yeild CPU
Middle: receiving event
High: yeild CPU again
Low: trying to lock
Low: locked
Low: sending event
High: trying to lock
Low: event sent
High: locked
High: unlocked
Middle: event received
Low: unlocked

System is going to be down
```

图 21.29

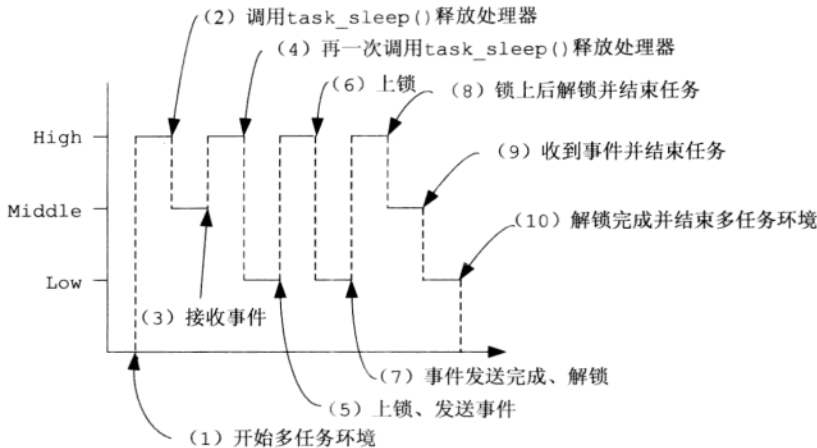


图 21.30

将这个时序图与图 21.26 比较我们可以发现，差别从第 6 步开始。该图中的第 6 步，由于高优先级的任务对低优先级的任务所持有的锁进行上锁操作，其造成低优先级的任务因为优先级继承而获得与高优先级的任务一样的优先级。因此，在高优先级的任务因为等待而放弃处理器后，低优先级的任务马上获得处理器继续运行。在第 7 步，当低优先级的任务事件发送完成并解锁后其优先级将恢复到原值，并触发任务调度。后面，高、中、低优先级的任务依次获得处理器运行。很明显，采用了优先级继承后中优先级的任务是在高优先级的任务之后才获得运行来处理收到的事件的。

在嵌入式软件开发中，我们需要避免发生任务反转的情形，否则很容易出现意想不到的结果。著名的“火星探路者”就曾因为没有将一个互斥锁设置为启用优先级继承功能，而频繁出现复位。对于这一事件的分析，读者可以参考随书光盘中 Project/embedded/docs 目录下的《NASA Mars Pathfinder Incident》一文。

21.2.5　递归锁

如果互斥锁被同一任务进行两次上锁，会出现什么问题呢？先看一看 deadlock 示例程序的运行结果，其源代码如图 21.31 所示。其中在第 36 和 39 行对同一个锁进行了两次上锁操作。

```
00026: #include "main.h"
00027: #include "device.h"
00028: #include "mutex.h"
00029: #include "console.h"
00030:
00031: static void task_deadlock (const char _name [], void *_p_arg)
00032: {
00033:     mutex_handle_t mutex = (mutex_handle_t) _p_arg;
00034:
00035:     console_print ("%s: going to take lock\n", _name);
```

```
00036:     (void) mutex_lock (mutex, WAIT_FOREVER);
00037:     console_print ("%s: lock is taken\n", _name);
00038:     console_print ("%s: going to take twice\n", _name);
00039:     (void) mutex_lock (mutex, WAIT_FOREVER);
00040:     multitasking_stop ();
00041: }
00042:
00043: error_t module_testapp (system_state_t _state)
00044: {
00045:     static task_handle_t task;
00046:     STACK_DECLARE (stack, 1024);
00047:     static mutex_handle_t mutex;
00048:     static device_handle_t ctrlc_handle;
00049:
00050:     if (STATE_INITIALIZING == _state) {
00051:         (void) mutex_create (&mutex, "Test");
00052:
00053:         (void) task_create (&task, "Task Low", 16, stack, sizeof (stack));
00054:         (void) task_start (task, task_deadlock, mutex);
00055:
00056:         (void) device_open (&ctrlc_handle, "/dev/ui/ctrlc", 0);
00057:     }
00058:     else if (STATE_DESTROYING == _state) {
00059:         (void) device_close (ctrlc_handle);
00060:         (void) task_delete (task);
00061:         (void) mutex_delete (mutex);
00062:     }
00063:     return 0;
00064: }
00065:
00066: int module_registration_entry (int argc, char *argv [])
00067: {
00068:     UNUSED (argc);
00069:     UNUSED (argv);
00070:
00071:     (void) module_register ("Interrupt", MODULE_INTERRUPT, CPU_LEVEL,
             module_interrupt);
00072:    (void) module_register ("Device", MODULE_DEVICE, DRIVER_LEVEL, module_device);
00073:     (void) module_register ("Timer", MODULE_TIMER, OS_LEVEL, module_timer);
00074:     (void) module_register ("Task", MODULE_TASK, OS_LEVEL, module_task);
00075:     (void) module_register ("Mutex", MODULE_MUTEX, OS_LEVEL, module_mutex);
00076:     (void) module_register ("TestApp", MODULE_TESTAPP, APPLICATION_LEVEL3,
             module_testapp);
00077:     return 0;
00078: }
```

图 21.31

图 21.32 示例说明了 deadlock 示例程序的运行结果——程序出现了死锁。原因很简单，现有互斥锁的实现会造成同一任务在第二次对互斥锁上锁时永久处于“等待”状态。

```
                       /embedded/build
$ make
                       /embedded/build
$ ./release/deadlock.exe

System is going to be up

Info: press Ctrl+C to terminate!
```

```
Task Low: going to take lock
Task Low: lock is taken
Task Low: going to take twice
```

图 21.32

在现实项目中，一个设计得好的软件模块不应当出现两次及以上的上锁情形。但是，当项目复杂且参与的人较多时，要做到这一点并不容易，因此我们需要从技术上解决这类问题。这正是引入递归锁这一概念的原因。递归锁允许一个任务对已持有的锁进行多次上锁操作而不会进入死锁状态。

为了实现递归锁，我们需要再一次对互斥锁的实现代码进行修改。修改点全部示例说明于图 21.33 中。

```
embedded/code/platform/sync/v3/inc/mutex.h
00044: typedef struct {
00045:     sync_object_t object_;
00046:     task_handle_t owner_;
00047:     bool inherited_;
00048:     task_priority_t original_;
00049:     bool is_recursive_;
00050:     usize_t reference_;
00051: } mutex_t, *mutex_handle_t;
00052:
embedded/code/platform/sync/v3/src/mutex.c
00036: static bool mutex_callback_lock (sync_object_handle_t _handle)
00037: {
00038:     mutex_handle_t p_mutex = (mutex_handle_t) _handle;
00039:     task_handle_t p_task = task_self ();
00040:
00041:     if (null == p_mutex->owner_) {
00042:         // grab the mutex
00043:         p_mutex->owner_ = p_task;
00044:         p_mutex->inherited_ = false;
00045:         p_mutex->reference_ = 0;
00046:         return true;
00047:     }
00048:     else if (p_mutex->is_recursive_ && p_task == p_mutex->owner_) {
00049:         p_mutex->reference_ ++;
00050:         return true;
00051:     }
00052:     return false;
00053: }
00054:
00069: static bool mutex_callback_unlock (sync_object_handle_t _handle,
00070:     error_t *_p_ecode)
00071: {
00072:     mutex_handle_t p_mutex = (mutex_handle_t) _handle;
00073:     task_handle_t p_task = task_self ();
00074:
00075:     // grab the mutex
00076:     if (is_in_interrupt () || is_invalid_task (p_task)) {
00077:         return ERROR_T (ERROR_MUTEX_INVCONTEXT);
00078:     }
00079:     if (p_mutex->owner_ != p_task) {
00080:         *_p_ecode = ERROR_T (ERROR_MUTEX_NOTOWNER);
```

```
00081:          return false;
00082:      }
00083:      if (p_mutex->reference_ > 0) {
00084:          p_mutex->reference_ --;
00085:          return true;
00086:      }
00087:      // restore owner's original priority
00088:      if (p_mutex->inherited_) {
00089:          p_mutex->inherited_ = false;
00090:          task_priority_change (p_mutex->owner_, p_mutex->original_);
00091:      }
00092:      if (task_bitmap_is_empty (&p_mutex->object_.pending_bitmap_)) {
00093:          // no task is pending on this semaphore
00094:          p_mutex->owner_ = null;
00095:          return true;
00096:      }
00097:      return false;
00098: }
00099:
00144: error_t mutex_create (mutex_handle_t *_p_handle, const char _name [],
00145:      bool _recursive)
00146: {
00147:      static bool initialized = false;
00148:      interrupt_level_t level;
00149:      error_t ecode;
00150:
00151:      level = global_interrupt_disable ();
00152:      if (!initialized) {
00153:          mutex_init ();
00154:          initialized = true;
00155:      }
00156:      global_interrupt_enable (level);
00157:      ecode = sync_object_alloc (&g_mutex_container,
00158:      (sync_object_handle_t *) _p_handle, _name);
00159:      if (0 == ecode) {
00160:          (*_p_handle)->owner_ = null;
00161:          (*_p_handle)->inherited_ = false;
00162:          (*_p_handle)->is_recursive_ = _recursive;
00163:      }
00164:      return ecode;
00165: }
```

图 21.33

在 mutex.h 中对互斥锁的管理数据结构增加了两个新的变量 is_recursive_和 reference_，分别用于记录该互斥锁是否使能了递归功能和记录锁持有者对其进行了多少次上锁操作。

mutex_callback_lock()函数的第 41～47 行是互斥锁第一次上锁时的实现，第一次上锁时需要对变量 reference_计数进行清 0。第 48～51 行允许对使能了递归功能的锁进行再一次上锁操作，在第 50 行直接返回 true 表示任务获得了锁。mutex_callback_unlock()函数的新增内容位于第 83～86 行。当 reference_的值大于 0 时，只需对计数减一并返回 true 表示解锁完成。针对 mutex_create()函数的改动，在第 145 行为函数增加了一个额外的参数，以表示所创建的互斥锁是否支持递归功能，在第 162 行将参数值保存起来。

图 21.34 显示了 recursive 示例程序的运行结果。recursive 的源代码与 deadlock 是完全一样的，只是所链接的库是 libsyncv3.a。从结果来看，并没有因为对同一锁进行多次上锁而造成死锁问题。

```
                    /embedded/build
$ make
                    /embedded/build
$ ./release/recursive.exe

System is going to be up

Info: press Ctrl+C to terminate!
Task Low: going to take lock
Task Low: lock is taken
Task Low: going to take twice

System is going to be down
```

图 21.34

21.3　事件

在 ClearRTOS 中，使用事件（event）不用象信号量和互斥锁那样，需要先调用创建函数进行创建。只要是任务，就具备了接收事件的能力。

21.3.1　应用场合

某个任务需要收到多个不同的通知后才做出反应，在这种情形下，运用信号量和互斥锁就不适用了，而是需要使用到另一种同步通信方式——事件。

事件被用于通知任务，其可以来源于其他任务或中断服务程序。与信号量和互斥锁不同的是，事件支持多个同时等待，且等待方式可以是“其中某个或多个事件”或“全部事件”。

21.3.2　程序实现

在 ClearRTOS 中，一个事件是用一个比特来表示的。由于保存事件的数据类型 event_set_t 被定义成位宽为 32（图 21.35 中的第 38 行），因此一个任务可以最多同时接收 32 个不同的事件。

```
00037: #ifndef __event_set_defined__
00038: typedef u32_t event_set_t, *event_set_handle_t;
00039: #define __event_set_defined__
00040: #endif
00041:
00042: #ifndef __event_option_defined__
00043: typedef u32_t event_option_t;
00044: #define __event_option_defined__
00045: #endif
00046:
00047: #define EVENT_WAIT_ALL           0x01
00048: #define EVENT_WAIT_ANY           0x02
00049: #define EVENT_RETURN_ALL         0x04
00050: #define EVENT_RETURN_EXPECTED    0x08
```

图 21.35

第 43 行的 event_option_t 数据类型用于指示一个任务对所期望的事件做出何种反应，其值定义在第 47～50 行。各值的含义如下：

- EVENT_WAIT_ALL：表示 event_receive()函数等待所有的所期望事件都收到后才返回，否则一直使任务处于“等待”状态(这里指没有设置超时值的情形)。
- EVENT_WAIT_ANY：表示 event_receive()函数只要收到任何一个(或多个)所期望的事件就返回。
- EVENT_RETURN_ALL：表示当 event_receive()函数返回时，将所有收到的事件都返回，其中可能包括所期望的和不期望的。
- EVENT_RETURN_EXPECTED：表示 event_receive()函数返回时，只返回所期望的事件。

为了让任务支持事件，需要对任务管理数据结构做一定的修改，如图 21.36 所示。

```
embedded/code/platform/task/v3/inc/task.h
00080: struct type_task {
00081:     ......
00103:     // for supporting event
00104:     event_set_t event_expected_;
00105:     event_set_t event_received_;
00106:     event_option_t event_option_;
00107: };
```

图 21.36

第 104～106 行新增了三个变量。event_expected_用于记录任务所期望收到的事件；event_received_指任务已收到的事件；event_option_记录了任务接收事件的选项。

21.3.2.1 接收事件

一个任务如果希望接收事件，需要调用 event_receive()函数，该函数的实现如图 21.37 所示。

```
00034: error_t event_receive (event_set_t _expected, event_set_handle_t _received,
00035:     msecond_t _timeout, event_option_t _option)
00036: {
00037:     interrupt_level_t level;
00038:     event_option_t wait_option = EVENT_WAIT_ALL | EVENT_WAIT_ANY;
00039:     event_option_t return_option = EVENT_RETURN_ALL | EVENT_RETURN_EXPECTED;
00040:     task_handle_t p_task;
00041:
00042:     if (is_in_interrupt ()) {
00043:         return ERROR_T (ERROR_EVENT_RECV_INVCONTEXT);
00044:     }
00045:     if (0 == _expected) {
00046:         return 0;
00047:     }
00048:     if (null == _received) {
00049:         return ERROR_T (ERROR_EVENT_RECV_INVPTR);
00050:     }
```

```
00051:     if ((wait_option == (wait_option & _option)) ||
00052:         (0 == (int)(wait_option & _option)) ||
00053:         (return_option == (return_option & _option)) ||
00054:         (0 == (int)(return_option & _option))) {
00055:         return ERROR_T (ERROR_EVENT_RECV_INVOPT);
00056:     }
00057:
00058:     level = global_interrupt_disable ();
00059:     p_task = task_self ();
00060:     if (is_invalid_task (p_task)) {
00061:         global_interrupt_enable (level);
00062:         return ERROR_T (ERROR_EVENT_RECV_INVCONTEXT);
00063:     }
00064: again:
00065:     // if expected event(s) is/are available, return it
00066:     if (EVENT_WAIT_ALL == (_option & EVENT_WAIT_ALL)) {
00067:         if ((_expected & p_task->event_received_) == _expected) {
00068:             if (EVENT_RETURN_ALL == (_option & EVENT_RETURN_ALL)) {
00069:                 *_received = p_task->event_received_;
00070:             }
00071:             else {
00072:                 *_received = _expected;
00073:                 p_task->event_received_ &= ~(*_received);
00074:             }
00075:             global_interrupt_enable (level);
00076:             return 0;
00077:         }
00078:     }
00079:     else {
00080:         if ((_expected & p_task->event_received_) != 0) {
00081:             if (EVENT_RETURN_ALL == (_option & EVENT_RETURN_ALL)) {
00082:                 *_received = p_task->event_received_;
00083:             }
00084:             else {
00085:                 *_received = _expected & p_task->event_received_;
00086:                 p_task->event_received_ &= ~(*_received);
00087:             }
00088:             global_interrupt_enable (level);
00089:             return 0;
00090:         }
00091:     }
00092:     // run here it means we need to block the task
00093:     p_task->timeout_ = _timeout;
00094:     (void) task_state_change (p_task, TASK_STATE_WAITING);
00095:     p_task->ecode_ = 0;
00096:     p_task->event_expected_ = _expected;
00097:     p_task->event_option_ = _option;
00098:     global_interrupt_enable (level);
00099:     task_schedule (null);
00100:     level = global_interrupt_disable ();
00101:     p_task->event_option_ = (event_option_t) 0;
00102:     if (0 == p_task->ecode_) {
00103:         // event(s) has/have received and need to return the event(s) expected
00104:         goto again;
00105:     }
00106:     global_interrupt_enable (level);
00107:     //lint -e(650)
00108:     if (ERROR_TASK_WAIT_TIMEOUT == MODULE_ERROR (p_task->ecode_)) {
```

```
00109:         p_task->ecode_ = ERROR_T (ERROR_EVENT_RECV_TIMEOUT);
00110:     }
00111:     return p_task->ecode_;
00112: }
```

图 21.37

_expected 参数用于指示任务所期望收到的事件；_received 参数用于存储 event_receive()函数返回收到的事件；_timeout 指示 event_recieve()函数接收事件的超时值，如果将其设置为 WAIT_FOREVER，则表示永远等待；_option 变量用于指示 event_receive()函数如何处理事件。

第 42～56 行是防止函数在中断状态下被调用，以及检查输入的参数是否合法。第 60～63 行检查函数是否是被任务所调用的，如果不是则返回错误。

当 event_receive()函数被调用时，需要先查看任务句柄中是否留存有所希望收到的事件。第 66～78 行代码是处理任务希望收到所有事件，第 79～91 行则对应于任务希望收到任意事件，这两部分代码请读者自行阅读。

如果程序运行到了第 93 行，则说明任务所期望的事件并没有（全部）收到，因此需要让任务进入等待状态。第 93 行记录任务的超时值。第 94 行调用 task_state_change()函数让任务进入等待状态。第 95 行清除任务的错误码。第 96 和 97 行保存任务所期望的事件和对事件的处理选项。第 99 行调用 task_schedule()函数触发一次任务调度。

每当任务所期望收到的事件满足任务的要求或者等待出现超时时，就会被唤醒，在这种情形下第 99 行的 task_schedule()函数将会返回，并继续执行其后的代码。第 101 行对 event_option_ 变量置 0，表示任务并不处于等待状态，在 event_send()函数（图 21.38）中需要用它进行判断以决定是否需要唤醒任务。第 102 行检查在等待事件的过程中是否出现了错误，如果没有则跳转到第 64 行继续执行；如果出现了错误，则在第 108 行判断是否是等待超时，如果是，则需要将错误码替换为 ERROR_EVENT_RECV_TIMEOUT，以便正确地反映超时是发生在事件管理模块的。

21.3.2.2 发送事件

调用 event_send()函数可以向指定的任务发送一个（或多个）事件，event_send()函数的实现如图 21.38 所示。其第一个参数指定事件是发送给哪一个任务的，而第二个参数指明所需发送的事件。

```
00114: error_t event_send (task_handle_t _handle, event_set_t _sent)
00115: {
00116:     interrupt_level_t level;
00117:     bool wakeup_needed = false;
00118:
00119:     level = global_interrupt_disable ();
00120:     if (is_invalid_task (_handle)) {
00121:         global_interrupt_enable (level);
00122:         return ERROR_T (ERROR_EVENT_RECV_INVRECEIVER);
```

```
00123:     }
00124:     _handle->event_received_ |= _sent;
00125:     if (0 == (int)_handle->event_option_) {
00126:         global_interrupt_enable (level);
00127:         return 0;
00128:     }
00129:     if (EVENT_WAIT_ALL == (_handle->event_option_ & EVENT_WAIT_ALL)) {
00130:         if ((_handle->event_expected_ & _handle->event_received_)
00131:             ==_handle->event_expected_) {
00132:             wakeup_needed = true;
00133:         }
00134:     }
00135:     else {
00136:         if ((_handle->event_expected_ & _handle->event_received_) != 0) {
00137:             wakeup_needed = true;
00138:         }
00139:     }
00140:     if (!wakeup_needed) {
00141:         global_interrupt_enable (level);
00142:         return 0;
00143:     }
00144:     (void) task_state_change (_handle, TASK_STATE_READY);
00145:     global_interrupt_enable (level);
00146:     task_schedule (null);
00147:     return 0;
00148: }
```

图 21.38

第 120～123 行检查事件的接收者是否是有效的，如果无效则返回错误。第 124 行将_sent 参数中的事件放入到接收任务的 event_received_变量中。第 125 行检查接收事件的任务是否处于等待状态，event_option_不为 0 表示处于等待状态。当事件的接收者并不处于等待状态时，事件的发送工作就算完成了，因此在第 127 行立即返回。第 129～139 行检查接收者是否收到了所期望的事件，如果是则需要唤醒该任务。第 144 行将接收者的状态设置为“就绪”。最后，在第 146 行触发一次任务调度。

从 event_send()函数的实现来看，相同的事件如果没有被接收者取走，则可能被发送者所覆盖。这一特性应当不影响应用的实现，否则选择事件作为通信方式就是一种错误。

21.3.2.3　清除事件

event_clear()函数可以由任务自己调用，以便在特定的情形下将收到的事件清空。图 21.39 是 event_clear()函数的实现代码，其中的第 160 行将任务用于记录收到事件的 event_received_ 变量置为 0。

```
00150: error_t event_clear ()
00151: {
00152:     interrupt_level_t level;
00153:     task_handle_t p_task = task_self ();
00154: 
00155:     level = global_interrupt_disable ();
00156:     if (is_invalid_task (p_task)) {
```

```
00157:         global_interrupt_enable (level);
00158:         return ERROR_T (ERROR_EVENT_CLEAR_INVCONTEXT);
00159:     }
00160:     p_task->event_received_ = 0;
00161:     global_interrupt_enable (level);
00162:     return 0;
00163: }
```

图 21.39

21.3.3 event 示例程序

事件的具体含义是由各任务根据应用程序的需要自行定义的，每个任务最多可以定义 32 个事件。event 示例程序要做的第一步是定义自己的事件，在图 21.40 的第 31 和 32 行分别定义了 EVENT_1 和 EVENT_2 两个事件。需要注意，在现实的项目中事件名称应定义得更具可读性。示例程序将创建三个任务，通过在它们之间发送事件的方式测试事件这一功能，其运行结果如图 21.41 所示。

```
00026: #include "main.h"
00027: #include "device.h"
00028: #include "event.h"
00029: #include "console.h"
00030:
00031: #define EVENT_1          0x01
00032: #define EVENT_2          0x02
00033:
00034: task_handle_t g_task1;
00035: task_handle_t g_task2;
00036: task_handle_t g_task3;
00037:
00038: static void task_task1 (const char _name [], void *_p_arg)
00039: {
00040:     event_set_t received;
00041:
00042:     UNUSED (_p_arg);
00043:
00044:     console_print ("%s: going to receive EVENT_1 or EVENT_2\n", _name);
00045:     (void) event_receive (EVENT_1 | EVENT_2, &received, WAIT_FOREVER,
00046:         EVENT_WAIT_ANY | EVENT_RETURN_EXPECTED);
00047:     console_print ("%s: EVENT_1 or EVENT_2 (%x) received\n", _name, received);
00048:     console_print ("%s: going to send EVENT_2\n", _name);
00049:     (void) event_send (g_task2, EVENT_2);
00050:     console_print ("%s: EVENT_2 is sent\n", _name);
00051: }
00052:
00053: static void task_task2 (const char _name [], void *_p_arg)
00054: {
00055:     event_set_t received;
00056:
00057:     UNUSED (_p_arg);
00058:
00059:     console_print ("%s: going to send EVENT_1\n", _name);
00060:     (void) event_send (g_task1, EVENT_1);
00061:     console_print ("%s: EVENT_1 is sent\n", _name);
00062:     console_print ("%s: going to receive EVENT_1 and EVENT_2\n", _name);
00063:     (void) event_receive (EVENT_1 | EVENT_2, &received, WAIT_FOREVER,
```

```
00064:          EVENT_WAIT_ALL | EVENT_RETURN_EXPECTED);
00065:      console_print ("%s: EVENT_1 and EVENT_2 (%x) received\n", _name, received);
00066: }
00067:
00068: static void task_task3 (const char _name [], void *_p_arg)
00069: {
00070:     UNUSED (_p_arg);
00071:
00072:     console_print ("%s: going to send EVENT_1\n", _name);
00073:     (void) event_send (g_task2, EVENT_1);
00074:     console_print ("%s: EVENT_1 is sent\n", _name);
00075:
00076:     multitasking_stop ();
00077: }
00078:
00079: error_t module_testapp (system_state_t _state)
00080: {
00081:     STACK_DECLARE (stack_for_task1, 1024);
00082:     STACK_DECLARE (stack_for_task2, 1024);
00083:     STACK_DECLARE (stack_for_task3, 1024);
00084:     static device_handle_t ctrlc_handle;
00085:
00086:     if (STATE_INITIALIZING == _state) {
00087:         (void) task_create (&g_task1, "Task1", 16, stack_for_task1,
00088:             sizeof (stack_for_task1));
00089:         (void) task_start (g_task1, task_task1, 0);
00090:
00091:         (void) task_create (&g_task2, "Task2", 11, stack_for_task2,
00092:             sizeof (stack_for_task2));
00093:         (void) task_start (g_task2, task_task2, 0);
00094:
00095:         (void) task_create (&g_task3, "Task3", 20, stack_for_task3,
00096:             sizeof (stack_for_task3));
00097:         (void) task_start (g_task3, task_task3, 0);
00098:
00099:         (void) device_open (&ctrlc_handle, "/dev/ui/ctrlc", 0);
00100:     }
00101:     else if (STATE_DESTROYING == _state) {
00102:         (void) device_close (ctrlc_handle);
00103:         (void) task_delete (g_task3);
00104:         (void) task_delete (g_task2);
00105:         (void) task_delete (g_task1);
00106:     }
00107:     return 0;
00108: }
00109:
00110: int module_registration_entry (int argc, char *argv [])
00111: {
00112:     UNUSED (argc);
00113:     UNUSED (argv);
00114:
00115:     (void) module_register ("Interrupt", MODULE_INTERRUPT, CPU_LEVEL,
            module_interrupt);
00116:     (void) module_register ("Device", MODULE_DEVICE, DRIVER_LEVEL, module_device);
00117:     (void) module_register ("Timer", MODULE_TIMER, OS_LEVEL, module_timer);
00118:     (void) module_register ("Task", MODULE_TASK, OS_LEVEL, module_task);
00119:     (void) module_register ("TestApp", MODULE_TESTAPP, APPLICATION_LEVEL3,
            module_testapp);
00120:     return 0;
00121: }
```

图 21.40

```
                    /embedded/build
$ make
                    /embedded/build
$ ./release/event.exe

System is going to be up

Info: press Ctrl-C to terminate!
Task2: going to send EVENT 1
Task2: EVENT 1 is sent
Task2: going to receive EVENT 1 and EVENT 2
Task1: going to receive EVENT 1 or EVENT 2
Task1: EVENT 1 or EVENT 2 (1) received
Task1: going to send EVENT 2
Task1: EVENT 2 is sent
Task3: going to send EVENT 1.
Task2: EVENT 1 and EVENT 2 (3) received
Task3: EVENT 1 is sent

System is going to be down
```

图 21.41

21.4　消息队列

消息队列是另一种在嵌入式软件开发中常用的、任务与任务间或中断与任务间的通信方法。该方法具有很强的灵活性和可定制性。

21.4.1　应用场合

在日常生活中，我们可能通过传纸条的方式实现多人之间的通信。纸条一旦由某人发起后，后面的人就可以通过阅读获取一定的信息，纸条所包含信息的多少完全取决于纸条中的内容，或许纸条上写了六条八卦新闻。如果纸条的阅读者愿意，可以拿起笔再写上一条新的八卦新闻，并将纸条继续传递下去。

嵌入式软件开发中的消息队列通信方式与上面的例子很相似，只不过将人换成了任务，纸条变成了消息。消息从软件的角度来看是一块内存，内存中的数据组织格式是事先定好的。当然，如果任务需要向消息内写入“八卦新闻”的话，得遵照之前定义好的格式进行书写，不然其他的任务会因为理解不了而“崩溃”。

使用消息队列进行通信有什么好处呢？

第一，因为消息用于携带信息进行传递，这就省去了定义全局变量。

第二，使用消息通信可以使任务之间去除一定的耦合性。通过以消息为接口，各相关任务不用太关心其他任务的具体实现，而只用关心消息的格式，以及自己应当针对消息做什么。正是因为运用消息能降低任务间的耦合度，所以在这样的设计中替换一个任务会相对简单，只要保证消息接口不变就行了。这为软件重构带来了一定的好处。

第三，通过使用消息的方式，将应用的业务逻辑分解成由多个任务去完成，堆栈溢出等问题就更容易定位。

21.4.2　程序实现

消息队列的实现将通过组合使用先进先出（First In First Out，FIFO）队列和信号量的方式实现。先进先出队列将提供消息的存储空间，而信号量将帮助实现任务同步功能。这种组合使用方式体现了代码复用的思想。

21.4.2.1　实现先进先出队列

先进先出队列其实是一块内存空间。将该内存空间分成多个“格子”，每个“格子”的大小是在创建队列时指定的，队列中“格子”数量也在创建时指定。在该节我们将“格子”称为队列元素，或元素，以区别于下节介绍消息队列时的消息。先进先出队列的管理数据结构如图 21.42 所示。

```
00033: #define FIFO_BUFFER_DECLARE(_name, _element_size, _capacity) \
00034:     static byte_t _name [_element_size * _capacity]
00035:
00036: typedef struct {
00037:     // how many elements the FIFO can contain
00038:     usize_t capacity_;
00039:     // buffer for containing elements
00040:     byte_t *buffer_addr_start_;
00041:     byte_t *buffer_addr_end_;
00042:     // element size
00043:     usize_t element_size_;
00044:     // how many elements are put into the FIFO
00045:     usize_t count_;
00046:     // position for getting next element
00047:     byte_t *cursor_get_;
00048:     // position for putting element
00049:     byte_t *cursor_put_;
00050: } fifo_t, *fifo_handle_t;
```

图 21.42

第 33 行定义的宏用于帮助分配队列所需的内存空间。其中第一个参数是指内存空间的变量名；第二个参数指定元素占据的空间大小；第三个参数指定队列中一共有多少个元素。在第 34 行通过定义静态数组的方式获得内存空间。

第 36～50 行定义的 fifo_t 数据结构用于管理先进先出队列。第 38 行的 capacity_变量用于记录队列中能存放多少个元素。第 40 和 41 行分别用于记录队列所占内存的起始地址和结束地址。第 43 行的 element_size_用于记录元素所占内存字节数。第 45 行的 count_变量用于记录队列中已存放了多少个元素。当要从队列中存取元素时，第 47 和 49 行的两个变量分别用于标识应当从哪一个“格子”中取和存。

1．队列初始化

队列的初始化需要通过调用 fifo_init()函数完成，其实现示例说明于图 21.43 中。

```
00029: void fifo_init (fifo_handle_t _handle, void *_buffer, usize_t _element_size,
00030:     usize_t _capacity)
00031: {
00032:     memset (_handle, 0, sizeof (*_handle));
00033:     memset (_buffer, 0, _element_size * _capacity);
00034:     _handle->capacity_ = _capacity;
00035:     _handle->buffer_addr_start_ = _buffer;
00036:     _handle->buffer_addr_end_ = _handle->buffer_addr_start_;
00037:     _handle->buffer_addr_end_ += _element_size * _capacity;
00038:     _handle->element_size_ = _element_size;
00039:     _handle->count_ = 0;
00040:     _handle->cursor_get_ = _buffer;
00041:     _handle->cursor_put_ = _buffer;
00042: }
```

图 21.43

注意：这个函数的功能是初始化，而不是分配。在调用 fifo_init()函数之前，需要先使用 FIFO_BUFFER_DECLARE 宏为队列准备好所需的存储内存空间。第二个参数正是用于指定使用 FIFO_BUFFER_DECLARE 宏所分配的内存的。第 32 和 33 行分别使用 memset()函数对句柄和队列空间进行初始化。第 34～38 行记录队列的相关参数。第 39 行将队列中的元素数目设置为 0。第 40 和 41 行分别设置对元素的取存光标位置。由于队列在初始化时为空，所以两个光标都设置为队列内存的开始地址。

2．存入元素

向先进先出队列中存入元素是通过调用 fifo_element_put()函数来完成的，其实现如图 21.44 所示。

```
00044: void fifo_element_put (fifo_handle_t _handle, const void *_p_element)
00045: {
00046:     if (_handle->element_size_ == sizeof (int)) {
00047:         // there is a assumption that the address of buffer_addr_start_
00048:         // is aligned with the size of int *
00049:         *(int *)(void *)_handle->cursor_put_ = *(int *)(void *)_p_element;
00050:     }
00051:     else if (_handle->element_size_ == sizeof (short)) {
00052:         *(short *)(void *)_handle->cursor_put_ = *(short *)(void *)_p_element;
00053:     }
00054:     else if (_handle->element_size_ == sizeof (char)) {
00055:         *(char *)_handle->cursor_put_ = *(char *)_p_element;
00056:     }
00057:     else {
00058:         memcpy (_handle->cursor_put_, _p_element, _handle->element_size_);
00059:     }
00060:     _handle->count_ ++;
00061:     _handle->cursor_put_ += _handle->element_size_;
00062:     if (_handle->cursor_put_ >= _handle->buffer_addr_end_) {
00063:         _handle->cursor_put_ = _handle->buffer_addr_start_;
00064:     }
00065: }
```

图 21.44

第 46～59 行通过判断元素的大小以决定如何将元素拷贝到队列的内存中，这样做是为了尽可能地避免使用 memcpy()函数。第 60 行将计数变量加一表示队列中多了一个元素。第 61～64 代码调整下一个元素在队列中的存放位置，并在光标到达末端时做回转调整。

注意：fifo_put()函数的实现并没有考虑队列已满这种情形。这里假设调用者在调用之前会通过 fifo_is_full()函数了解队列是否仍有存储空间。

3. 取出元素

从队列中取出一个元素需要使用 fifo_element_get()函数，其实现如图 21.45 所示。

```
00067: void fifo_element_get (fifo_handle_t _handle, void *_p_element)
00068: {
00069:     void *p_element = _handle->cursor_get_;
00070:     _handle->count_ --;
00071:     _handle->cursor_get_ += _handle->element_size_;
00072:     if (_handle->cursor_get_ >= _handle->buffer_addr_end_) {
00073:         _handle->cursor_get_ = _handle->buffer_addr_start_;
00074:     }
00075:     if (_handle->element_size_ == sizeof (int)) {
00076:         // there is a assumption that the address of buffer_addr_start_
00077:         // is aligned with the size of int *
00078:         *(int *)_p_element = *(int *)p_element;
00079:     }
00080:     else if (_handle->element_size_ == sizeof (char)) {
00081:         *(char *)_p_element = *(char *)p_element;
00082:     }
00083:     else {
00084:         memcpy (_p_element, p_element, _handle->element_size_);
00085:     }
00086: }
```

图 21.45

第 69 行先记录所取元素在队列中的位置。接着第 70～74 行调整下一次的存取位置，并做光标回转处理。第 72～85 行将队列中的元素内容拷贝到函数的第二个参数所指向的内存中。

同样地，fifo_element_get()函数的实现并没有考虑取元素时队列是否为空这一情形。

4. 其他辅助函数

图 21.46 列出了操作队列的其他辅助函数的实现。实现很简单，在此不做更多解释。

```
00064: static inline bool fifo_is_empty (const fifo_handle_t _handle)
00065: {
00066:     return (bool) (0 == _handle->count_);
00067: }
00068:
00069: static inline bool fifo_is_full (const fifo_handle_t _handle)
00070: {
00071:     return (bool) (_handle->count_ == _handle->capacity_);
```

```
00072: }
00073:
00074: static inline usize_t fifo_capacity (const fifo_handle_t _handle)
00075: {
00076:     return _handle->capacity_;
00077: }
00078:
00079: static inline usize_t fifo_count (const fifo_handle_t _handle)
00080: {
00081:     return _handle->count_;
00082: }
00083:
00084: static inline usize_t fifo_element_size (const fifo_handle_t _handle)
00085: {
00086:     return _handle->element_size_;
00087: }
```

图 21.46

21.4.3 实现消息队列

有了先进先出队列的实现后，消息队列的实现就相对简单了。所需的数据结构定义如图 21.47 所示。

```
00033: #define QUEUE_BUFFER_DECLARE(_name, _element_size, _capacity) \
00034:     FIFO_BUFFER_DECLARE(_name, _element_size, _capacity)
00035:
00036: typedef struct {
00037:     dll_node_t node_;
00038:     semaphore_handle_t semaphore_;
00039:     magic_number_t magic_number_;
00040:     fifo_t fifo_;
00041:     char name_ [NAME_MAX_LENGTH + 1];
00042: } queue_t, *queue_handle_t;
```

图 21.47

第 33 行定义的宏用于为消息队列分配内存空间，为消息队列重新定义这个宏可向使用者屏蔽消息队列是由先进先出队列实现的这一细节。第 37 行的 node_变量用做链表结点。第 38 行是消息队列所需用到的信号量。第 39 行 magic_number_变量的目的相信读者已经很熟悉了。第 40 行是先进先出队列所需的管理数据。最后，在第 41 行定义了用于存放消息队列名字的内存空间。

图 21.48 列出了其他的模块变量。第 35 行定义了用于标识有效消息队列的标识值。第 37～38 行所定义的宏用于检查消息队列的有效性。第 41～44 行定义了用于记录模块统计信息的数据结构。第 46 行是消息队列池，它是一个数组，数组元素的个数由 CONFIG_MAX_QUEUE 宏所决定。第 47 行是模块的统计变量。第 48 和 49 行的两个链表分别用于存储空闲的和正被使用的消息队列。

```
00033: #define CONFIG_MAX_QUEUE        8
00034: #define QUEUE_LAST_INDEX        (CONFIG_MAX_QUEUE - 1)
00035: #define MAGIC_NUMBER_QUEUE      0x51554555L
00036:
00037: #define is_invalid_handle(_handle) \
00038:     ((_handle == null) || ((_handle)->magic_number_ != MAGIC_NUMBER_QUEUE))
00039:
00040:
00041: typedef struct {
00042:     statistic_t no_queue_;
00043:     statistic_t message_lost_;
00044: } queue_statistic_t;
00045:
00046: static queue_t g_queue_pool [CONFIG_MAX_QUEUE];
00047: queue_statistic_t g_statistics;
00048: static dll_t g_free_queue;
00049: static dll_t g_used_queue;
```

图 21.48

21.4.3.1　创建队列

队列创建函数 queue_create()的实现如图 21.49 所示。

```
00075: static void queue_init ()
00076: {
00077:     usize_t idx;
00078:
00079:     for (idx = 0; idx <= QUEUE_LAST_INDEX; ++ idx) {
00080:         dll_push_tail (&g_free_queue, &g_queue_pool [idx].node_);
00081:     }
00082: }
00083:
00084: error_t queue_create (const char _name [],queue_handle_t *_p_handle,
00085:     void *_buffer, usize_t _element_size, usize_t _capacity)
00086: {
00087:     static bool initialized = false;
00088:     interrupt_level_t level;
00089:     queue_handle_t handle;
00090:     #define SEMAPHORE_PREFIX "Queue:"
00091:     char sem_name [NAME_MAX_LENGTH + 1] = SEMAPHORE_PREFIX;
00092:     error_t ecode;
00093:
00094:     if (is_in_interrupt ()) {
00095:         return ERROR_T (ERROR_QUEUE_CREATE_INVCONTEXT);
00096:     }
00097:
00098:     level = global_interrupt_disable ();
00099:     if (!initialized) {
00100:         queue_init ();
00101:         initialized = true;
00102:     }
00103:     handle = (queue_handle_t)dll_pop_head (&g_free_queue);
00104:     if (0 == handle) {
00105:         g_statistics.no_queue_ ++;
00106:         global_interrupt_enable (level);
00107:         return ERROR_T (ERROR_QUEUE_CREATE_NOQUE);
```

```
00108:     }
00109:     memset (handle, 0, sizeof (*handle));
00110:     handle->magic_number_ = MAGIC_NUMBER_QUEUE;
00111:     global_interrupt_enable (level);
00112:     strncpy (&sem_name [sizeof (SEMAPHORE_PREFIX)], _name,
00113:         sizeof (sem_name) - (sizeof (SEMAPHORE_PREFIX) + 1));
00114:     sem_name [sizeof (sem_name) - 1] = 0;
00115:     ecode = semaphore_create (&handle->semaphore_, _name, 0);
00116:     if (ecode != 0) {
00117:         goto error;
00118:     }
00119:     //lint -e{774}
00120:     if (null == _name) {
00121:         handle->name_ [0] = 0;
00122:     }
00123:     else {
00124:         strncpy (handle->name_, _name, sizeof (handle->name_) - 1);
00125:         handle->name_ [sizeof (handle->name_) - 1] = 0;
00126:     }
00127:     fifo_init (&handle->fifo_, _buffer, _element_size, _capacity);
00128:     dll_push_tail (&g_used_queue, &handle->node_);
00129:     *_p_handle = handle;
00130:     return 0;
00131: error:
00132:     dll_push_head (&g_free_queue, &handle->node_);
00133:     *_p_handle = null;
00134:     return ecode;
00135: }
```

图 21.49

第 99～102 行检查模块是否已初始化，如果没有则调用 queue_init()函数进行初始化。queue_init()的实现位于第 75～82 行，它将每一个队列放入到空闲链表 g_free_queue 中。第 103 行从空闲链表中获取一个队列。第 104～108 行处理无队列可被分配的情形。在没有队列可分配的情形下，在第 105 行更新相应的统计信息，接着在第 107 行返回错误。

第 109 行对整个队列数据结构进行置 0 初始化。第 110 行设置队列的标识值。第 112～113 行是为要创建的信号量准备名称。第 115 行为队列创建信号量，且将信号量的初始值设置为 0，表示队列中没有消息。第 120～126 行是处理队列的名称。第 127 行调用 fifo_init()函数对先进先出队列进行初始化。第 128 行将被分配出的队列放入使用链表 g_used_queue 中。第 129 行将队列返回给函数的调用者。

21.4.3.2　删除队列

队列删除函数 queue_delete()的实现如图 21.50 所示。

```
00137: error_t queue_delete (queue_handle_t _handle)
00138: {
00139:     interrupt_level_t level;
00140: 
00141:     if (is_in_interrupt () && STATE_UP == system_state ()) {
00142:         return ERROR_T (ERROR_QUEUE_DELETE_INVCONTEXT);
00143:     }
```

```
00144:
00145:     if (is_invalid_handle (_handle)) {
00146:         return ERROR_T (ERROR_QUEUE_DELETE_INVHANDLE);
00147:     }
00148:
00149:     level = global_interrupt_disable ();
00150:     _handle->magic_number_ = 0;
00151:     dll_remove (&g_used_queue, &_handle->node_);
00152:     dll_push_tail (&g_free_queue, &_handle->node_);
00153:     global_interrupt_enable (level);
00154:
00155:     (void) semaphore_delete (_handle->semaphore_);
00156:     return 0;
00157: }
```

图 21.50

第 150 行将标识置成 0 使消息队列无效。第 151 行将队列从使用链表中删除，并在第 152 行将其加入空闲队列中。第 155 行释放队列的信号量。

21.4.3.3　发送消息

当一个任务或者中断服务程序要发送消息时，需调用 queue_message_send()函数，该函数的实现列于图 21.51 中。

```
00159: error_t queue_message_send (queue_handle_t _handle, const void *_p_element)
00160: {
00161:     interrupt_level_t level;
00162:
00163:     if (is_invalid_handle (_handle)) {
00164:         return ERROR_T (ERROR_QUEUE_SEND_INVHANDLE);
00165:     }
00166:
00167:     level = global_interrupt_disable ();
00168:     if (fifo_is_full (&_handle->fifo_)) {
00169:         g_statistics.message_lost_ ++;
00170:         global_interrupt_enable (level);
00171:         return ERROR_T (ERROR_QUEUE_SEND_FULL);
00172:     }
00173:     fifo_element_put (&_handle->fifo_, _p_element);
00174:     global_interrupt_enable (level);
00175:     return semaphore_give (_handle->semaphore_);
00176: }
```

图 21.51

其中的第 168～172 行是处理队列已满的情形。第 173 行将消息作为一个元素存入先进先出队列中。第 175 行更新信号量的计数。

21.4.3.4　接收消息

图 21.52 是消息接收函数的实现代码。

```
00178: error_t queue_message_receive (queue_handle_t _handle, msecond_t _timeout,
```

```
00179:     void *_p_element)
00180: {
00181:     error_t ecode;
00182:     interrupt_level_t level;
00183:
00184:     if (is_invalid_handle (_handle)) {
00185:        return ERROR_T (ERROR_QUEUE_RECV_INVHANDLE);
00186:     }
00187:
00188:     ecode = semaphore_take (_handle->semaphore_, _timeout);
00189:     if (ecode != 0) {
00190:        return ecode;
00191:     }
00192:     level = global_interrupt_disable ();
00193:     fifo_element_get (&_handle->fifo_, _p_element);
00194:     global_interrupt_enable (level);
00195:     return 0;
00196: }
```

图 21.52

在第 188 行通过调用 semaphore_take()函数查看队列中是否有消息。在没有消息的情形下，这一调用会导致调用任务进入“等待”状态。如果 semaphore_take()函数返回 0 值，则说明消息队列中存在消息，于是在第 193 行从先进先出队列中取出消息返回给调用者。

21.4.3.5 状态查询

消息队列是否是空的或满的，可以通过调用 queue_is_empty()和 queue_is_full()函数进行查询，两个函数的实现都非常简单，如图 21.53 所示。

```
00198: bool queue_is_empty (const queue_handle_t _handle)
00199: {
00200:     interrupt_level_t level;
00201:     bool is_empty;
00202:
00203:     if (is_invalid_handle (_handle)) {
00204:        return ERROR_T (ERROR_QUEUE_EMPTY_INVHANDLE);
00205:     }
00206:
00207:     level = global_interrupt_disable ();
00208:     is_empty = fifo_is_empty (&_handle->fifo_);
00209:     global_interrupt_enable (level);
00210:     return is_empty;
00211: }
00212:
00213: bool queue_is_full (const queue_handle_t _handle)
00214: {
00215:     interrupt_level_t level;
00216:     bool is_full;
00217:
00218:     if (is_invalid_handle (_handle)) {
00219:        return ERROR_T (ERROR_QUEUE_FULL_INVHANDLE);
00220:     }
00221:
00222:     level = global_interrupt_disable ();
00223:     is_full = fifo_is_full (&_handle->fifo_);
```

```
00224:     global_interrupt_enable (level);
00225:     return is_full;
00226: }
```

图 21.53

21.4.3.6　模块管理

queue_dump()函数可用于查看系统中所有消息队列的状态，其实现如图 21.54 所示。

```
00228: static bool queue_dump_for_each (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00229: {
00230:     queue_handle_t handle = (queue_handle_t)_p_node;
00231:
00232:     UNUSED (_p_dll);
00233:     UNUSED (_p_arg);
00234:
00235:     console_print (" Name: %s\n", handle->name_);
00236:     console_print (" Message Size: %u\n", fifo_element_size (&handle->fifo_));
00237:     console_print (" Capacity: %u\n", fifo_capacity (&handle->fifo_));
00238:     console_print (" Filled: %u\n", fifo_count (&handle->fifo_));
00239:     console_print ("\n");
00240:     return true;
00241: }
00242:
00243: void queue_dump ()
00244: {
00245:     if (is_in_interrupt ()) {
00246:         return;
00247:     }
00248:
00249:     scheduler_lock ();
00250:     console_print ("\n\n");
00251:     console_print ("Summary\n");
00252:     console_print ("-------\n");
00253:     console_print (" Supported: %u\n", CONFIG_MAX_QUEUE);
00254:     console_print (" Allocated: %u\n", dll_size (&g_used_queue));
00255:     console_print (" .BSS Used: %u\n", ((address_t)&g_used_queue
00256:         - (address_t)g_queue_pool) + sizeof (g_used_queue));
00257:     console_print ("\n");
00258:     console_print ("Statistics\n");
00259:     console_print ("----------\n");
00260:     console_print (" No Object: %u\n", g_statistics.no_queue_);
00261:     console_print (" Messsage Lost: %u\n", g_statistics.message_lost_);
00262:     console_print ("\n");
00263:     console_print ("Queue Details\n");
00264:     console_print ("-------------\n");
00265:     (void) dll_traverse (&g_used_queue, queue_dump_for_each, 0);
00266:     console_print ("\n");
00267:     scheduler_unlock ();
00268: }
```

图 21.54

图 21.55 是消息队列模块回调函数 module_queue()的实现。它同样只关心当系统关闭时，是否存在消息队列没有回收的现象。另外，如果在系统关闭时消息队列并不为空，则会打印出告警信息。

```
00051: static bool queue_check_for_each (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00052: {
00053:     queue_handle_t handle = (queue_handle_t)_p_node;
00054:
00055:     UNUSED (_p_dll);
00056:     UNUSED (_p_arg);
00057:
00058:     if (!queue_is_empty (handle)) {
00059:         console_print ("Warning: queue \"%s\" isn't empty\n", handle->name_);
00060:     }
00061:     console_print ("Error: queue \"%s\" isn't deleted\n", handle->name_);
00062:     return true;
00063: }
00064:
00065: error_t module_queue (system_state_t _state)
00066: {
00067:     if (STATE_DESTROYING == _state) {
00068:         // check whether all queues created have been deleted or not,
00069:         // if not take them as error
00070:         (void) dll_traverse (&g_used_queue, queue_check_for_each, 0);
00071:     }
00072:     return 0;
00073: }
```

图 21.55

21.4.4 queue 示例程序

图 21.56 是 queue 示例程序的源程序。这个示例程序创建了一个消息大小为一个字节、最多可存放 20 个的消息队列。其中一个任务将一个字符串当做消息发送给另一个任务。接收任务在收到发过来的消息后，将其打印出来。程序的运行结果如图 21.57 所示。

```
00026: #include "main.h"
00027: #include "device.h"
00028: #include "queue.h"
00029: #include "console.h"
00030:
00031: #define CONFIG_MAX_ELEMENT      20
00032:
00033: static task_handle_t g_task_producer;
00034: static task_handle_t g_task_consumer;
00035:
00036: STACK_DECLARE (g_stack_for_producer, 1024);
00037: STACK_DECLARE (g_stack_for_consumer, 1024);
00038:
00039: QUEUE_BUFFER_DECLARE (g_queue_buffer, sizeof (char), CONFIG_MAX_ELEMENT);
00040: static queue_handle_t g_queue;
00041:
00042: static void task_producer (const char _name [], void *_p_arg)
00043: {
00044:     queue_handle_t p_queue = (queue_handle_t)_p_arg;
00045:     char greeting [] = "Have a good day!\n";
00046:     char *p_char = greeting;
00047:     int len = sizeof (greeting) - 1;
00048:
```

```
00049:     UNUSED (_name);
00050:
00051:     while (len -- > 0) {
00052:         (void) queue_message_send (p_queue, p_char ++);
00053:     }
00054: }
00055:
00056: static void task_consumer (const char _name [], void *_p_arg)
00057: {
00058:     queue_handle_t p_queue = (queue_handle_t)_p_arg;
00059:     char ch;
00060:
00061:     UNUSED (_name);
00062:
00063:     while (!queue_is_empty (p_queue)) {
00064:         (void) queue_message_receive (p_queue, WAIT_FOREVER, &ch);
00065:         console_print ("%c", ch);
00066:     }
00067:
00068:     queue_dump ();
00069:     multitasking_stop ();
00070: }
00071:
00072: error_t module_testapp (system_state_t _state)
00073: {
00074:     static device_handle_t ctrlc_handle;
00075:
00076:     if (STATE_INITIALIZING == _state) {
00077:         (void) queue_create ("Test", &g_queue, g_queue_buffer,
00078:             sizeof (char), CONFIG_MAX_ELEMENT);
00079:
00080:         (void) task_create (&g_task_producer, "Producer", 11,
00081:             g_stack_for_producer, sizeof (g_stack_for_producer));
00082:         (void) task_start (g_task_producer, task_producer, g_queue);
00083:
00084:         (void) task_create (&g_task_consumer, "Consumer", 13,
00085:             g_stack_for_consumer, sizeof (g_stack_for_consumer));
00086:         (void) task_start (g_task_consumer, task_consumer, g_queue);
00087:
00088:         (void) device_open (&ctrlc_handle, "/dev/ui/ctrlc", 0);
00089:     }
00090:     else if (STATE_DESTROYING == _state) {
00091:         (void) device_close (ctrlc_handle);
00092:         (void) task_delete (g_task_consumer);
00093:         (void) task_delete (g_task_producer);
00094:         (void) queue_delete (g_queue);
00095:     }
00096:     return 0;
00097: }
00098:
00099: int module_registration_entry (int argc, char *argv [])
00100: {
00101:     UNUSED (argc);
00102:     UNUSED (argv);
00103:
00104:     (void) module_register ("Interrupt", MODULE_INTERRUPT, CPU_LEVEL,
            module_interrupt);
00105:     (void) module_register ("Device", MODULE_DEVICE, DRIVER_LEVEL, module_device);
00106:     (void) module_register ("Timer", MODULE_TIMER, OS_LEVEL, module_timer);
00107:     (void) module_register ("Task", MODULE_TASK, OS_LEVEL, module_task);
```

```
00108:     (void) module_register ("Queue", MODULE_QUEUE, OS_LEVEL, module_queue);
00109:     (void) module_register ("TestApp", MODULE_TESTAPP, APPLICATION_LEVEL3,
               module_testapp);
00110:     return 0;
00111: }
```

图 21.56

```
                        /embedded/build
$ make
                        /embedded/build
$ ./release/queue.exe

System is going to be up

Info: press Ctrl+C to terminate!
Have a good day!


Summary
-------
  Supported: 8
  Allocated: 1
  .BSS Used: 508

Statistics
----------
     No Object: 0
  Messsage Lost: 0

Queue Details
-------------
  Name: Test
    Message Size: 1
         Capacity: 20
             Count: 0

System is going to be down
```

图 21.57

21.4.5 使用指南

使用消息队列需要考虑存放消息内容的内存来源问题，有两种做法。一是像 queue 示例程序那样将消息内容直接存放在消息队列中。示例中的消息内容所占内存大小是一个字节，但完全可以根据需要定义成某一数据结构的大小。

这种方法的优点是省去了为存放消息内容的内存进行动态分配，这可以避免内存泄漏问题的发生。但会带来性能问题，即消息收发时存在内存拷贝。

二是将消息内容间接保存在消息队列中。保存消息内容所需的内存通过调用 malloc()这样的函数动态获得，然后将内存块的地址存入消息队列中。显然，消息的发送者需要负责分配内存，而接收者要承担释放内存的责任。

这种方法的好处是明显的，消息在传递时很高效。但缺点也很明显，需要很小心地防范内

存泄漏问题。使用这种方法时如果需要缓解频繁的内存分配与释放操作所带来的处理器开销，可以考虑采用内存池的方式加以弥补（参见 22.2 节）。

两种方法需要根据系统的特点加以权衡后做出选择，并没有明确的答案。

21.5　死锁及预防

任务同步的目的是为了消除系统的竞争问题，以及做到任务间流畅、有序地协同工作。但是任务同步还很容易带来另一个问题——死锁。

死锁问题在讲解互斥锁时有所涉及，当时的场景是因为任务对同一互斥锁进行重复的上锁而造成的，最终的解决方法是为互斥锁增加了递归功能。

死锁发生的另一种场景是由多个任务需要同时持有多个同步对象。比如，存在 A 和 B 两个任务，且任务 A 和 B 在某种场合下需要同时持有 X 和 Y 两个同步对象。如果任务 A 持有了 X 同步对象且试图持有 Y 同步对象，但是任务 B 却持有了 Y 同步对象并试图持有 X 同步对象。在这种情形下就会出现死锁。

造成死锁的根本原因，是由多个任务对同步对象的获取顺序不一致而造成的。如果所有的任务在需要持有多个同步对象时总是采用相同的获取顺序，就一定不会出现死锁问题。读者可以对上面的死锁情形采用获取顺序一致的方法进行分析，看看是不是解决了死锁问题。

除了任务采用一致的同步对象获取顺序外，使用同步对象时注意以下几点将有助于减少死锁问题的发生。

- 力争做到同步对象的使用具有很强的局部性。将同步对象的获取与释放两个动作总是放在同一个函数中实现，而不是将两个动作分散在不同的两个函数中。
- 减小同步对象所保护资源的粒度。这存在两个层面的意思：第一，同步对象的获取时间点应尽可能靠近代码的同步点；第二，如果同步对象所保护的资源太多，存在多个任务需要分别使用不同的部分，则采用将大资源分割成多个小资源的方法，再用多个同步对象分别对它们进行管理。
- 同步对象的使用如果有造成死锁的趋势，则说明软件的模块化设计存在概念不清的问题。在这种情形下，应致力于分析概念不清是如何造成的，并通过调整模块的结构消除其中的不清。一个概念清晰的模块能有效地消除同步对象的使用混乱问题，从而避免死锁问题。

21.6　小结

信号量是一种资源计数器，用于记录一种资源的多个实例。它除了可以被用于任务与任务

之间的通信手段外，还可以被运用于中断与任务间的通信手段。

互斥锁体现的是排他精神。互斥锁只能用做任务与任务之间的同步。如果需要实现任务与中断间的互斥处理，应当通过中断控制这一途径。为了解决优先级反转问题，互斥锁大多有优先级继承功能。递归互斥锁能解决任务多次上锁造成的死锁问题。

事件可被运用于同步多个通知事件的发生，它同样可以作为中断与任务间的通信手段。

消息队列具有更大的灵活性，可以通过自定义消息的方式，使得任务与任务以及中断与任务间进行自由的通信。

练习与思考

1. 为什么不允许 semaphore_take()函数在中断状态被调用？

2. 信号量是否也要解决像互斥锁那样的递归调用问题？

3. 在使用消息队列进行通信的系统中，有时会出现因为消息队列暂时满而造成后续消息的丢失。如果丢失的消息中有的又非常重要，那如何解决这类问题？

4. 如果一个任务需要从两个消息队列中接收消息，请通过本章介绍的任务同步与通信手段设计一种实现方法。

5. 死锁的发生，往往是因为对同步对象的获取顺序不一致而造成的。尽管我们知道需要通过保证一致的获取顺序来避免出现死锁问题，但如何将这种保证通过软件设计来实现呢？



第 22 章 内存管理，协调动态内存的使用

动态内存是通过调用函数而获得的，来源可以是堆或内存池。当从堆中获得内存时，所需内存大小通过函数的参数加以指定，因此从堆中获得的内存大小是非固定的。与之不同的是，当从内存池中获取内存时，所获得内存的大小是固定的，其数值在内存池初始化时就决定了。内存管理的目的就是实现动态内存管理并提供操控动态内存的接口函数。

与动态内存相对应的是静态内存，其通过定义全局或静态（数组）变量的方式获得，无须调用函数。因此，静态内存的一大优势是无须专门对其进行管理。

本章将分别介绍一种堆管理和内存池管理实现，以帮助读者理解内存管理到底是什么。内存分配算法有很多种，读者可以参照专门介绍操作系统的书籍来掌握更多的内存分配算法。不论是怎样的内存管理算法，最终都得在效率与内存管理模块自身所占用的资源之间进行平衡。

22.1 堆管理

这里将要介绍的堆管理算法，作者给它取名为 mblock 算法，这一名字来源于算法的数据结构名。在实现这个内存管理算法时，将通过三个不同版本的演进使其功能愈加完善。

在讲解堆程序之前，让我们先看一看 heapv1 示例程序，以便了解堆管理模块的功能。

22.1.1 heapv1 示例程序

在这个示例程序中，除了调用 heap_alloc()和 heap_free()函数进行内存分配与释放外，还通过不断地调用 heap_dump()函数以了解模块所管理内存的状况，其源程序如图 22.1 所示。

```
00026: #include "main.h"
00027: #include "device.h"
00028: #include "heap.h"
00029: #include "console.h"
00030:
```

```
00031: static void task_test (const char _name [], void *_p_arg)
00032: {
00033:     void *p_buf1, *p_buf2, *p_buf3;
00034:     error_t result;
00035:
00036:     UNUSED (_name);
00037:     UNUSED (_p_arg);
00038:
00039:     console_print ("\n");
00040:     console_print ("===========\n");
00041:     console_print ("Before Test ->\n");
00042:     console_print ("===========\n");
00043:     heap_dump ();
00044:
00045:     console_print ("=============================\n");
00046:     console_print ("Test Case 1): allocate 1 byte ->\n");
00047:     console_print ("=============================\n");
00048:     p_buf1 = heap_alloc (1);
00049:     console_print (" Allocated Addr: %p\n", p_buf1);
00050:     heap_dump ();
00051:
00052:     console_print ("================================\n");
00053:     console_print ("Test Case 2): allocate 32K bytes ->\n");
00054:     console_print ("================================\n");
00055:     p_buf2 = heap_alloc (32*1024);
00056:     console_print (" Allocated Addr: %p\n", p_buf2);
00057:     heap_dump ();
00058:
00059:     console_print ("================================\n");
00060:     console_print ("Test Case 3): allocate 64K bytes ->\n");
00061:     console_print ("================================\n");
00062:     p_buf3 = heap_alloc (64*1024);
00063:     console_print (" Allocated Addr: %p\n", p_buf3);
00064:     heap_dump ();
00065:
00066:     console_print ("============================\n");
00067:     console_print ("Test Case 4): free 32K bytes ->\n");
00068:     console_print ("============================\n");
00069:     if ((result = heap_free (p_buf2, 32*1024)) != 0) {
00070:         console_print ("heap_free () returns %s!\n", errstr (result));
00071:     }
00072:     console_print (" Freed Addr: %p\n", p_buf2);
00073:     heap_dump ();
00074:
00075:     console_print ("======================================\n");
00076:     console_print ("Test Case 5): allocate 64K bytes again ->\n");
00077:     console_print ("======================================\n");
00078:     p_buf2 = heap_alloc (64*1024);
00079:     console_print (" Allocated Addr: %p\n", p_buf2);
00080:     heap_dump ();
00081:
00082:     console_print ("=========================\n");
00083:     console_print ("Test Case 6): free 1 byte ->\n");
00084:     console_print ("=========================\n");
00085:     if ((result = heap_free (p_buf1, 1)) != 0) {
00086:         console_print ("heap_free () returns %s!\n", errstr (result));
00087:     }
00088:     console_print (" Freed Addr: %p\n", p_buf1);
00089:     heap_dump ();
00090:
00091:     console_print ("============================\n");
00092:     console_print ("Test Case 7): free 64K bytes ->\n");
00093:     console_print ("============================\n");
```

```
00094:    if ((result = heap_free (p_buf2, 64*1024)) != 0) {
00095:       console_print ("heap_free () returns %s!\n", errstr (result));
00096:    }
00097:    console_print (" Freed Addr: %p\n", p_buf2);
00098:    heap_dump ();
00099:
00100:    console_print ("=============================\n");
00101:    console_print ("Test Case 8): free 64K bytes ->\n");
00102:    console_print ("=============================\n");
00103:    if ((result = heap_free (p_buf3, 64*1024)) != 0) {
00104:       console_print ("heap_free () returns %s!\n", errstr (result));
00105:    }
00106:    console_print (" Freed Addr: %p\n", p_buf3);
00107:    heap_dump ();
00108:    multitasking_stop ();
00109: }
00110:
00111: error_t module_testapp (system_state_t _state)
00112: {
00113:    static task_handle_t handle;
00114:    STACK_DECLARE (stack, 1024);
00115:
00116:    if (STATE_INITIALIZING == _state) {
00117:       (void) task_create (&handle, "Test", 16, stack, sizeof (stack));
00118:       (void) task_start (handle, task_test, 0);
00119:    }
00120:    else if (STATE_DESTROYING == _state) {
00121:       (void) task_delete (handle);
00122:    }
00123:    return 0;
00124: }
00125:
00126: int module_registration_entry (int argc, char *argv [])
00127: {
00128:    UNUSED (argc);
00129:    UNUSED (argv);
00130:
00131:    (void) module_register ("Interrupt", MODULE_INTERRUPT, CPU_LEVEL,
              module_interrupt);
00132:    (void) module_register ("Device", MODULE_DEVICE, DRIVER_LEVEL, module_device);
00133:    (void) module_register ("Timer", MODULE_TIMER, OS_LEVEL, module_timer);
00134:    (void) module_register ("Task", MODULE_TASK, OS_LEVEL, module_task);
00135:    (void) module_register ("Heap", MODULE_HEAP, OS_LEVEL, module_heap);
00136:    (void) module_register ("TestApp", MODULE_TESTAPP, APPLICATION_LEVEL3,
              module_testapp);
00137:    return 0;
00138: }
```

图 22.1

其中，我们需要重点关注 task_test()函数中的 8 个测试用例。对于每一个测试用例，通过调用 heap_dump()函数了解堆的状态。图 22.2 列出了 heapv1 示例程序的最终运行结果，后面我们将对照这些结果来讲解 mblock 算法的具体实现。读者所获得的运行结果中的地址值与本书中的可能不同，但并不妨碍理解 mblock 的实现。

```
$ make
```

```
$ ./release/heapv1.exe

System is going to be up

===========
Before Test ->
===========

Free Block(s)
-------------
  Block Start Addr: 7e6e0008, size: 1000000 (16777216)

Summary
-------
  Alignment: 8
  Heap Addr: 7e6e0008
       Total: 1000000 (16777216)
        Used: 0 (0)
        Free: 1000000 (16777216)

==============================
Test Case 1): allocate 1 byte ->
==============================
    Allocated Addr: 0x7e6e0008

Free Block(s)
-------------
  Block Start Addr: 7e6e0010, size: ffffff8 (16777208)

Summary
-------
  Alignment: 8
  Heap Addr: 7e6e0008
       Total: 1000000 (16777216)
        Used: 8 (8)
        Free: ffffff8 (16777208)

==============================
Test Case 2): allocate 32K bytes ->
==============================
    Allocated Addr: [illegible]

Free Block(s)
-------------
  Block Start Addr: [illegible], size: ff7ff8 (1674444[illegible])

Summary
-------
  Alignment: 8
  Heap Addr: 7e6e0008
       Total: 1000000 (16777216)
        Used: 8008 (32776)
        Free: ff7ff8 ([illegible]744440)

==============================
Test Case 3): allocate [illegible] bytes ->
==============================
    Allocated Addr: [illegible]

Free Block(s)
```

```
-------------
  Block Start Addr: 7e6f8010, size: fe7ff8 (16678904)

Summary
-------
  Alignment: 8
  Heap Addr: 7e6e0008
       Total: 1000000 (16777216)
        Used: 18008 (98312)
        Free: fe7ff8 (16678904)

============================
Test Case 4): free 32K bytes ->
============================
   Freed Addr: 0x7e6e0010

Free Block(s)
-------------
  Block Start Addr: 7e6e0010, size: 8000 (32768)
  Block Start Addr: 7e6f8010, size: fe7ff8 (16678904)

Summary
-------
  Alignment: 8
  Heap Addr: 7e6e0008
       Total: 1000000 (16777216)
        Used: 10008 (65544)
        Free: fefff8 (16711672)

============================
Test Case 5): allocate 64K bytes again ->
============================
   Allocated Addr: 0x7e6f8010

Free Block(s)
-------------
  Block Start Addr: 7e6e0010, size: 8000 (32768)
  Block Start Addr: 7e708010, size: fd7ff8 (16613368)

Summary
-------
  Alignment: 8
  Heap Addr: 7e6e0008
       Total: 1000000.(16777216)
        Used: 20008 (131080)
        Free: fdfff8 (16646136)

============================
Test Case 6): free 1 byte ->
============================
   Freed Addr: 0x7e6e0008

Free Block(s)
-------------
  Block Start Addr: 7e6e0008, size: 8008 (32776)
  Block Start Addr: 7e708010, size: fd7ff8 (16613368)

Summary
-------
  Alignment: 8
```

```
  Heap Addr: [illegible]
      Total: [illegible]
       Used: [illegible]
       Free: [illegible]

==========
Test Case 7: [illegible]
==========
    Freed Addr: [illegible]

Free Block(s):
--------------
  Block Start Addr: [illegible], size: [illegible]
  Block Start Addr: [illegible], size: [illegible]

Summary
-------
  Alignment: [illegible]
  Heap Addr: [illegible]
      Total: [illegible]
       Used: [illegible]
       Free: [illegible]

============
Test Case 8: [illegible]
============
    Freed Addr: [illegible]

Free Block(s):
--------------
  Block Start Addr: [illegible], size: [illegible]

Summary
-------
  Alignment: [illegible]
  Heap Addr: [illegible]
      Total: [illegible]
       Used: [illegible]
       Free: [illegible]


System is [illegible]
```

图 22.2

22.1.2　程序实现

图 22.3 示例说明了堆管理模块中将要使用到的基本数据类型和宏。ROUND_UP 和 ROUND_DOWN 两个宏都是用于对地址_addr 进行边界对齐处理的。ROUND_UP 宏可能使对齐后的地址值比_addr 更大，而 ROUND_DOWN 则相反。IS_ALIGNED 宏可用于判定_addr 是否是以_alignment 个字节对齐的。ALIGN 宏的功能与 ROUND_UP 是一样的。

```
00029: typedef unsigned int uint32_t;
00037: typedef uint32_t address_t;
00039: typedef uint32_t msize_t;
00041: typedef uint32_t usize_t;
```

```
00031: #define ROUND_UP(_addr, _alignment)
       (((_addr) + (_alignment - 1)) & ~(_alignment - 1))
00032: #define ROUND_DOWN(_addr, _alignment) ((_addr) & ~(_alignment - 1))
00033: #define IS_ALIGNED(_addr, _alignment) (((_addr) & (_alignment - 1)) == 0)
00034:
00035: #define ALIGN(_addr, _alignment) \
00036:     (((_addr) + ((_alignment) - 1)) & ~((_alignment) - 1))
```

图 22.3

22.1.2.1　模块初始化

堆管理模块的初始化需要调用 heap_init()函数，其程序实现如图 22.4 所示。

```
00033: typedef struct {
00034:     address_t next_;
00035:     msize_t size_;
00036: } mblock_t;
00037:
```

```
00033: static bool g_initialized;
00034: static mblock_t g_mblock_free;
00035: static address_t g_heap_addr_start;
00036: static address_t g_heap_addr_end;
00037: static msize_t g_heap_size;
00038: // minimal allocation size
00039: static msize_t g_min_alloc_size;
00040: // the alignment for each allocation:
00041: //     g_alignment_bytes = (1 << g_alignment_in_bits);
00042: static msize_t g_alignment_bytes;
00043: static msize_t g_alignment_in_bits;
00044:
00045: error_t heap_init (address_t _start, address_t _end, msize_t _alignment_in_bits)
00046: {
00047:     mblock_t* p_mblock;
00048:
00049:     if (_start > _end) {
00050:         return ERROR_T (ERROR_HEAP_INIT_INVADDR);
00051:     }
00052:     if (g_initialized) {
00053:         return 0;
00054:     }
00055:     g_alignment_in_bits = _alignment_in_bits;
00056:     g_alignment_bytes = ((msize_t)1 << g_alignment_in_bits);
00057:     if (sizeof (int) > g_alignment_bytes) {
00058:         return ERROR_T (ERROR_HEAP_INIT_INVALIGN);
00059:     }
00060:
00061:     // initialize the g_mblock_free, it holds first free block
00062:     g_mblock_free.next_ = ALIGN (_start, g_alignment_bytes);
00063:     // the length is in g_alignment_bytes unit
00064:     g_mblock_free.size_ = (_end - g_mblock_free.next_) >>
00065:         g_alignment_in_bits;
00066:     // initialize the free block and its size
00067:     p_mblock = (mblock_t *)g_mblock_free.next_;
00068:     p_mblock->next_ = 0;
00069:     p_mblock->size_ = g_mblock_free.size_;
00070:     // save the start and end address for later use
```

```
00071:     g_heap_addr_start = g_mblock_free.next_;
00072:     g_heap_addr_end = g_mblock_free.next_ +
00073:         (g_mblock_free.size_ << g_alignment_in_bits);
00074:     g_heap_size = g_mblock_free.size_;
00075:     // minimal allocation size should be bigger than the size of mblock_t
00076:     g_min_alloc_size = (sizeof (mblock_t) + g_alignment_bytes - 1) >>
00077:     g_alignment_in_bits;
00078:
00079:     g_initialized = true;
00080:     return 0;
00081: }
```

图 22.4

heap_init()函数的三个参数分别指明了被管理内存的开始和结束地址，以及所分配出来的内存地址应当满足多少字节的边界对齐。注意，第三个参数的指定形式不是字节数，而是比特位数。比如，如果希望所分配出来的内存地址满足 8 字节边界对齐，第三个参数应指定为 3。

heap.h 中的第 33～36 行定义 mblock_t 数据结构，其中 next_和 size_变量的作用需要通过图示的方法进行解释，后面马上涉及。

heap.c 中的第 55 行将函数的第三个参数保存到全局变量中，第 56 行将比特数转换为字节数。第 57～59 行确保所希望的边界对齐字节数不会小于 int 类型的大小。

第 62～69 行对 g_mblock_free 全局变量和堆上的第一个 mblock_t 数据结构进行初始化。图 22.5 是根据 heapv1 示例程序所获得的内存空间布局。

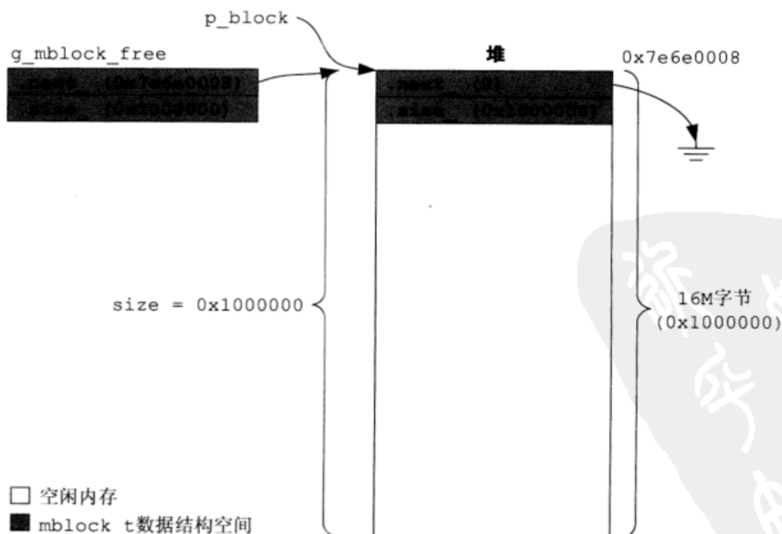


图 22.5[1]

[1] 图中 size_变量的大小是以 heapv1 程序的运行结果来表示的，是以字节而不是程序中以 g_alignment_bytes 变量的值为单位的。

从图中可以清楚地了解 mblock_t 数据结构的意义。对于 g_mblock_free 变量，mblock_t 数据结构中的 next_变量所表示的是堆空间中的第一块空闲内存空间，而其 size_变量记录的是整个堆空间有多少空闲内存可用，其单位不是字节，而是以 g_alignment_bytes 变量的值为单位的。位于堆空间中的 mblock_t 数据结构，被放置在空闲内存空间的开始处，其中 next_变量指向的是下一个空闲区块，而 size_指明结构所在空闲区块的大小，它同样以 g_alignment_bytes 变量的值为单位。由于在初始化时，堆是一块连续的空闲空间，所以位于最头上的 mblock_t 数据结构中的 next_的值为 0，表示其后没有空闲的内存块。

第 71～74 行将堆空间的位置和大小信息记录下来。第 76 行对 g_min_alloc_size 变量进行初始化，这个变量用于记录所分配出来的内存块的最小尺寸，它也是以 g_alignment_bytes 变量的值为单位的。从实现中可以看出，最小分配单位必须保证能放下一个 mblock_t 数据结构，这是因为当被分配出去的内存需要释放时，需要在其中放下一个 mblock_t 结构，以将其链入空闲内存块链 g_mblock_free 中。第 79 行将 g_initialized 变量设置为 true，表示模块已经被初始化过了。

图 22.1 中并不能看到 heap_init()函数是如何被调用的，这是因为堆管理模块也是通过注册回调函数的形式进行初始化的。其模块回调函数实现如图 22.6 所示。

```
/code/platform/memory/heap/v1/inc/heap.h
00038: typedef struct {
00039:     address_t start_;
00040:     address_t end_;
00041:     msize_t alignment_in_bits_;
00042: } heap_info_t;
00043:
```

```
/code/platform/memory/heap/v1/src/heap.c
00251: error_t module_heap (system_state_t _state)
00252: {
00253:     if (STATE_INITIALIZING == _state) {
00254:         heap_info_t heap_info;
00255:         heap_info_get (&heap_info);
00256:         return heap_init (heap_info.start_, heap_info.end_,
00257:             heap_info.alignment_in_bits_);
00258:     }
00259:     else if (STATE_DESTROYING == _state) {
00260:        msize_t size = ((g_heap_size - g_mblock_free.size_) << g_alignment_in_bits);
00261:         if (0 != size) {
00262:             console_print ("Error: %d bytes of memory are not freed");
00263:         }
00264:     }
00265:     return 0;
00266: }
```

图 22.6

先看一看系统初始化时的实现。heap.c 的第 254 行定义了一个类型为 heap_info_t 的局部变量，以便后面在第 255 行调用 heap_info_get()函数获得堆空间的起始地址和大小。heap_info_t 数据结构的定义可以从 heap.h 的第 38～42 行找到。一旦获得了堆空间信息后，第 256 行通过调用 heap_init()函数对堆管理模块进行初始化。

在系统终止化时，需要查看所管理的内存是否全部都释放了。如果没有，在第 262 行打印出一行错误日志。注意，第一版（程序中用 v1 目录加以表示）的实现中并没有记录每一次内存分配是在哪里发生的，所以只知道是否有内存未释放而不知具体点，在后续版本的实现中将改善这一问题。

heap_info_get()函数需根据每个系统去实现，图 22.7 示例说明了在 Linux 操作系统和 Cygwin 环境中的实现。

```
00083: #define SYSTEM_MEM_SIZE            (16*1024*1024)
00084: #define SYSTEM_ALIGNMENT_BITS     3
00085:
00086: void heap_info_get (heap_info_t *_p_info)
00087: {
00088:     void* p_heap = malloc (SYSTEM_MEM_SIZE);
00089:     if (0 == p_heap) {
00090:         console_print ("Error: cannot malloc for heap\n");
00091:         exit (-1);
00092:     }
00093:
00094:     _p_info->start_ = (address_t) p_heap;
00095:     _p_info->end_ = ((address_t)p_heap) + SYSTEM_MEM_SIZE;
00096:     _p_info->alignment_in_bits_ = SYSTEM_ALIGNMENT_BITS;
00097: }
```

图 22.7

第 83 行定义了堆空间的大小是 16 MB，第 83 行定义了边界对齐字节数为 8。在第 88 行通过调用 malloc()函数从实际的操作系统获得所需的空间供 ClearRTOS 使用。第 94～96 行对函数参数进行赋值。请注意，第 88 行所分配出来的内存并没有其他的地方对其释放，这一内存的释放操作是交给 Linux 操作系统去完成的。之所以不增加释放的代码，目的是使得 ClearRTOS 中的实现更接近现实的嵌入式系统。在一个真实的嵌入式系统中，并不需要通过调用 malloc()函数的形式获取内存，而是通过__bss__变量和系统配置。

22.1.2.2 分配内存

内存分配函数 heap_alloc()的实现如图 22.8 所示。

```
00083: void* heap_alloc (msize_t _size)
00084: {
00085:     register mblock_t *p_pre, *p_next, *p_superfluous;
00086:     msize_t size_required, size_superfluous;
00087:     interrupt_level_t level;
00088:
00089:     if (!g_initialized || is_in_interrupt () || 0 == _size) {
00090:         return null;
00091:     }
00092:
00093:     // convert the _size into g_alignment_bytes unit
00094:     size_required = ((_size + g_alignment_bytes - 1) >> g_alignment_in_bits);
```

```
00095:     if (g_min_alloc_size > size_required) {
00096:         size_required = g_min_alloc_size;
00097:     }
00098:     level = global_interrupt_disable ();
00099:     if (g_mblock_free.size_ < size_required) {
00100:         global_interrupt_enable (level);
00101:         return null;
00102:     }
00103:     p_pre = p_next = &g_mblock_free;
00104:     for (;;) {
00105:         p_next = (mblock_t *)p_next->next_;
00106:         if (0 == p_next) {
00107:             // till now we have traversed all the block and didn't find
00108:             // a block for this request
00109:             global_interrupt_enable (level);
00110:             return null;
00111:         }
00112:         // if this block size cannot meet the request, continue next block
00113:         if (p_next->size_ < size_required) {
00114:             p_pre = p_next;
00115:             continue;
00116:         }
00117:         // block size is bigger or equal to size_required, get the
00118:         // remaining free size
00119:         size_superfluous = p_next->size_ - size_required;
00120:         if (size_superfluous <= g_min_alloc_size) {
00121:             size_required = p_next->size_;
00122:             p_superfluous = (mblock_t *)p_next->next_;
00123:             break;
00124:         }
00125:         // put the superfluous block into the free list
00126:         p_superfluous = (mblock_t *)((address_t)p_next +
00127:             (size_required << g_alignment_in_bits));
00128:         p_superfluous->next_ = p_next->next_;
00129:         p_superfluous->size_ = size_superfluous;
00130:         break;
00131:     }
00132:     p_pre->next_ = (address_t)p_superfluous;
00133:     g_mblock_free.size_ -= size_required;
00134:     global_interrupt_enable (level);
00135:     return p_next;
00136: }
```

图 22.8

第 89～91 行做必要的检查。第 94～97 行计算出要分配的内存数量，是以 g_alignment_bytes 变量的值为单位的。如果所需分配的数量少于 g_min_alloc_size，则以 g_min_alloc_size 为准，原因在前面已经介绍了。

第 99～102 行是在分配之前先检查整个堆中的空闲内存是否够这次分配，这一检查有助于内存不够时快速返回。第 104～131 行通过遍历堆空间中的每块空闲区，以找到比所需的大或相等的第一块。第 106～111 行查看是否没有空闲块可用于分配，如果没有则返回 null 表示分配失败。第 113～116 行，如果正在遍历的空闲块比所需大小更小，则执行 continue 语句继续查看下一个空闲块。

程序运行到第 119 行，说明现在正在遍历的空闲块的大小满足所需分配的值。于是，可以从中“切”一块出来进行分配。第 119 行先计算出从空闲块中分配出所需大小后还剩多少。第 120 行看一下多余的内存是不是不足 g_min_alloc_size，如果是就不用“切”了，而是直接将整个块分配出去。第 126～129 行将多余出来的部分与空闲链串在一起。

在 heapv1 示例程序中，当运行完 1 至 3 的测试用例后，所获得的内存布局如图 22.9 所示。

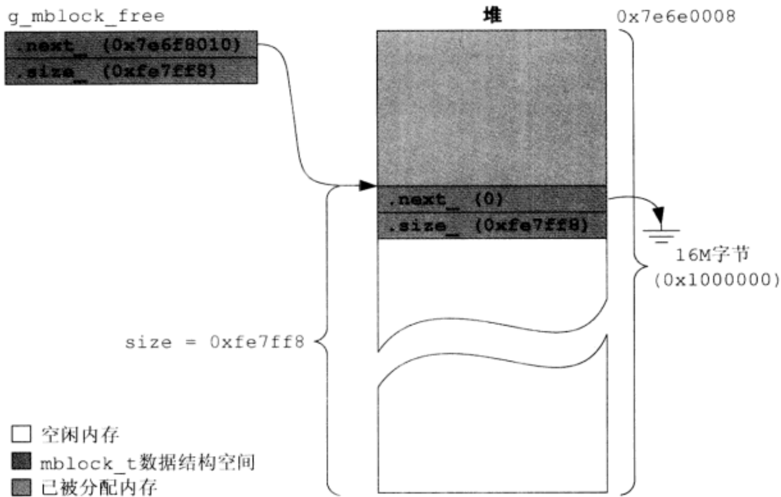


图 22.9

22.1.2.3 释放与合并内存

heapv1 示例程序中的测试用例 4 是第一次尝试释放一块内存，内存释放需要调用 heap_free() 函数，其实现如图 22.10 所示。这个函数的实现需要两个参数，其中的第二个参数用于指定所释放的字节数是多少，在后续的版本中将通过简化而省去这个参数。

```
platform/memory/heap/v1/src/heap.c
00138: error_t heap_free (void* _p_buf, msize_t _size)
00139: {
00140:     register address_t p_buf = (address_t)_p_buf;
00141:     register mblock_t* p_pre,*p_next;
00142:     interrupt_level_t level;
00143:     msize_t size;
00144:
00145:     if (!g_initialized) {
00146:         return ERROR_T (ERROR_HEAP_FREE_NOTINIT);
00147:     }
00148:     if (is_in_interrupt () && STATE_UP == system_state ()) {
00149:         return ERROR_T (ERROR_HEAP_ALLOC_INVCONTEXT);
00150:     }
00151:     if (0 == _size) {
00152:         return ERROR_T (ERROR_HEAP_FREE_INVSIZE);
00153:     }
00154:     if (0 == p_buf || ((p_buf & (g_alignment_bytes - 1)) != 0) ||
```

```
00155:         p_buf < g_heap_addr_start ||
00156:         p_buf > g_heap_addr_end || _size > g_heap_size) {
00157:         return ERROR_T (ERROR_HEAP_FREE_INVBUF);
00158:     }
00159:
00160:     size = ((_size + g_alignment_bytes - 1) >> g_alignment_in_bits);
00161:     if (g_min_alloc_size > size) {
00162:         size = g_min_alloc_size;
00163:     }
00164:     level = global_interrupt_disable ();
00165:     g_mblock_free.size_ += size;
00166:     if (0 == g_mblock_free.next_) {
00167:         g_mblock_free.next_ = p_buf ;
00168:         p_next = (mblock_t *)p_buf ;
00169:         p_next->size_ = size;
00170:         p_next->next_ = 0;
00171:         global_interrupt_enable (level);
00172:         return 0;
00173:     }
00174:     else {
00175:         p_pre = &g_mblock_free;
00176:         p_next = (mblock_t *)g_mblock_free.next_;
00177:     }
00178:     // find the right position of the list
00179:     while (p_pre->next_ < p_buf ) {
00180:         p_pre = p_next;
00181:         p_next = (mblock_t *)p_next->next_;
00182:         if (null == p_next) {
00183:             break;
00184:         }
00185:     }
00186:     // merge with the previously adjacent block if needed
00187:     if ((((p_pre->size_ << g_alignment_in_bits) + (address_t)p_pre)
00188:         == p_buf ) && (p_pre != &g_mblock_free)) {
00189:         p_pre->size_ += size;
00190:     }
00191:     else {
00192:         p_pre->next_ = p_buf ;
00193:         p_pre = (mblock_t *)p_buf;
00194:         p_pre->size_ = size;
00195:         p_pre->next_ = (address_t)p_next;
00196:     }
00197:     if (0 == p_next) {
00198:         // this is the last block no more mergence is needed
00199:         global_interrupt_enable (level);
00200:         return 0;
00201:     }
00202:     // merge with the following adjacent block if needed
00203:     if (((p_pre->size_ << g_alignment_in_bits) + (address_t)p_pre)
00204:         == (address_t)p_next) {
00205:         p_pre->size_ += p_next->size_;
00206:         p_pre->next_ = p_next->next_;
00207:     }
00208:     global_interrupt_enable (level);
00209:     return 0;
00210: }
```

图 22.10

第 145～158 行做释放前的必要检查。与 heap_alloc()类似的是，在第 160 行将_size 转换成以 g_alignment_bytes 为单位。第 161～163 行对所需释放的内存大小进行调整。第 165 行更新

整个堆空闲内存的数量。

第 166～173 行处理所释放的内存块是堆中的唯一空闲内存块的情形。其处理很简单，只需将空闲块简单地链到 g_mblock_free 链表中即可。如果所释放的内存块并不是整个堆中唯一的一块空闲块，则需要找到释放空闲块应插入的位置，这正是第 179～185 行的作用。查找工作是通过比较每一个空闲块的开始地址来实现的。

一旦找到插入位置的前一个空闲块，就需要看一看这个空闲块与被释放的块是否是紧挨着的。如果是，只要更新被找到的空闲块的大小就行了，这部分代码对应于第 187～190 行；否则，需要将被释放的块与找到的空闲块链起来，对应的代码是第 192～195 行。

将释放的块链入链表后，还没有完成释放操作。堆管理很重要的一个工作是需要进行内存合并，以减少内存碎片。第 187～190 行完成了被释放内存块与低地址空闲内存块的合并，接下来，还得尝试与高地址空闲内存块的合并。

第 197 行查看后面没有空闲内存块的情形，在这种情形下就不需要进行内存合并了；否则，在第 203 行需要通过计算看一看被释放的内存块与后面的空闲块是否是紧紧相连的。如果是，则将后面空闲块的大小加到前面的空闲块上就行了，且不再需要后一个空闲块的 mblock_t 数据结构。

heapv1 示例程序在运行完第 4 个测试用例后，将获得图 22.11 所示的内存布局。请读者重点关注各空闲块是如何通过 mblock_t 数据结构链在一起的。

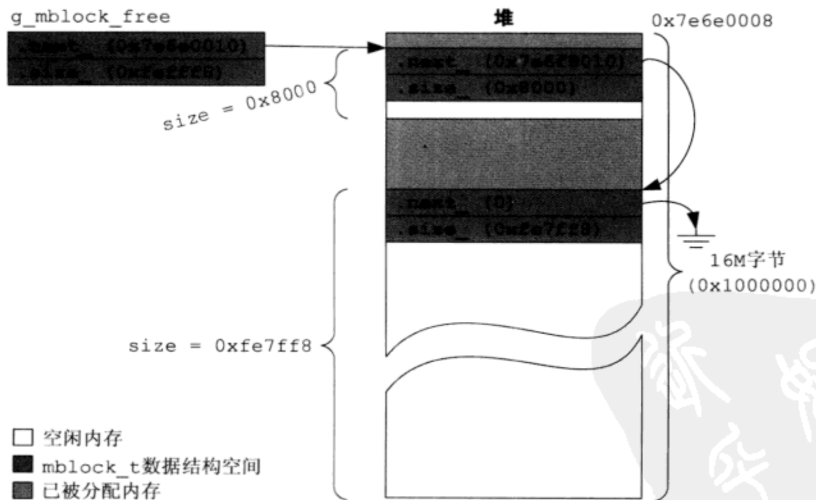


图 22.11

测试用例 5 是在测试用例 4 释放了 32KB 内存的情形下，再请求分配 64KB 的内存。此时，从图 22.11 可以看出第一块空闲内存块的大小是 32KB(0x8000)，所以只能从第二块空闲块中分配。运行完测试用例 5 后的内存布局如图 22.12 所示。

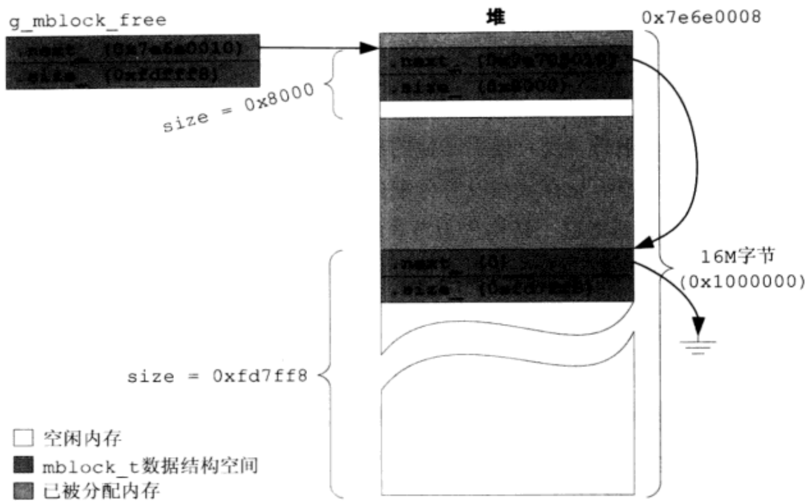


图 22.12

从测试用例 6 开始将释放所有从堆中分配出来的内存。在测试用例 6 中，当释放了第一次分配出来的 1 个字节后，内存布局如图 22.13 所示。从这次释放可以看出，第一次所请求分配的 1 个字节堆管理模块实际上却为之分配了 8 个字节。这次释放，也完成了一次与后续内存块的合并操作，这从图 22.13 中的第一块空闲块的大小从 0x8000 变成了 0x8008 可以看出。

可以想象，当运行完了 heapv1 示例程序中的测试用例 7 和 8 以后，我们将得到一个与图 22.5 完全一样的内存布局。

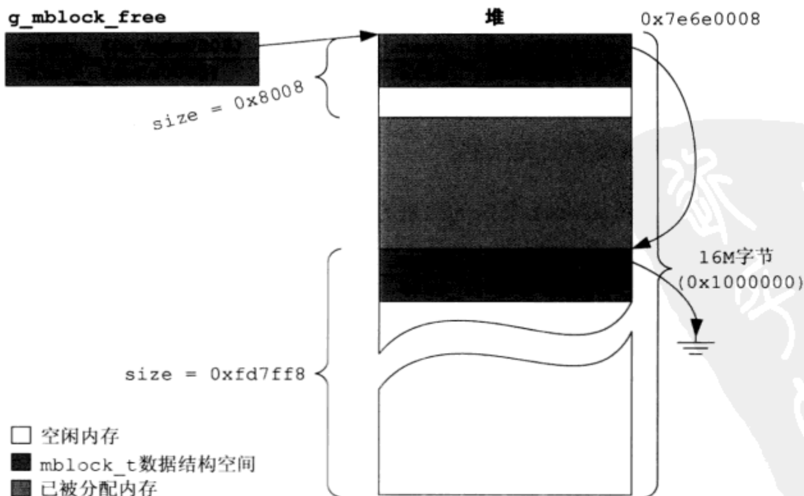


图 22.13

22.1.3　设计改进

下面让我们来分析一下 mblock 算法的优缺点，以便找到改进之处。第一个优点是，mblock 算法除了使用几个全局变量外，没有使用其他的额外内存用做管理。正如我们前面所看到的那样，其巧妙之处在于将管理信息存放在没有分配出去的空闲内存块中。通过在空闲块的头部获得一个 mblock_t 结构，然后将这个结构作为链表链接起来，达到管理所有空闲块的目的。这种方法虽然简单，但是后面我们将看到这也正是其缺点所在。

第二个优点是内存的合并[②]操作非常高效。从实现中可以看出，为了合并一块内存，只需要找到（这个找可并不高效）被释放内存的前后空闲块，然后通过简单的加法计算就能判断出与前后的空闲块是否是相连的。当需要做合并操作时，只需对相邻的 mblock_t 结构中的 size_ 进行更新就行了。

接着，让我们看一看 mblock 算法的缺点。由于内存管理信息是放置在空闲内存块的头部的，这就存在一种危险，就是当用户不小心对这个空闲块之前的用户内存块进行写操作并出现写溢出时，空闲块头部的 mblock_t 结构信息将会被篡改，这将导致该空闲内存块从系统中“消失”。此外，由于 mblock_t 中的内容被改写以后，其中的内容就没有意义了，如果还采信这一信息结果是可以想象的——系统将崩溃，且崩溃点位于内存管理模块内。

不论采用怎样的内存管理方法，管理信息都有可能因为程序的意外操作（比如对未初始化的指针进行写操作）而导致管理信息被破坏。当内存管理数据被意外破坏时，如果堆管理模块在使用这些信息之前能发现而不采信它，并终止后续的内存处理操作，就可以避免内存模块的崩溃（但还是会在其他点崩溃）。

另一个缺点是，释放内存时需要提供被释放内存的大小。当系统分配的内存块较多时这是一种负担。在使用 C 标准库中的 free()函数时，我们并不需要提供被释放内存的大小是多少，其大小信息是由内存管理模块自己记录的。

这两个缺点也指明了对 mblock 算法的改进方向。

22.1.3.1　管理数据的完整性保护

数据的完整性保护可以说是无处不在，我们经常使用的 TCP/IP 协议就是采用校验和的形式以保证数据包在各主机间可靠地传送。我们也可以借用这一思想来提高堆管理模块的鲁棒性。堆管理模块中最关键的管理信息源自 mblock_t 数据结构，对 mblock_t 数据结构进行一定更改后的实现如图 22.14 所示。

② 这里只指合并操作，而不隐含释放操作。

```
embedded/code/platform/memory/heap/v2/inc/heap.h
00033: typedef struct {
00034:     maddr_t next_;
00035:     maddr_t next_not_;
00036:     msize_t size_;
00037:     msize_t size_not_;
00038: } mblock_t;
```

图 22.14

第 35 和 37 行的成员变量是新增加的，从名称来看分别是指 next_和 size_取反后的数值。通过增加这两个变量，当设置好 next_和 size_两个变量的值之后，分别取反而获得 next_not_和 size_not_两个变量的值。在对 mblock_t 数据结构进行访问之前，就可以通过这两个变量检查信息是否是完整的。图 22.15 列出了为了实现 mblock_t 数据结构的完整性而增加的两个函数。

```
embedded/code/platform/memory/heap/v2/src/heap.c
00033: static const maddr_t ADDRESS_MARK = 0x24891513;
00034: static const msize_t SIZE_MARK = 0x18132A49;
00035:
00049: static inline void mblock_integrate (register mblock_t* _p_mblock)
00050: {
00051:     _p_mblock->next_not_ = ADDRESS_MARK ^ _p_mblock->next_;
00052:     _p_mblock->size_not_ = SIZE_MARK ^ _p_mblock->size_;
00053: }
00054:
00055: static inline bool mblock_integrity_check (register const mblock_t *_p_mblock)
00056: {
00057:     if (_p_mblock->next_not_ != (ADDRESS_MARK ^ _p_mblock->next_)) {
00058:         return false;
00059:     }
00060:     if (_p_mblock->size_not_ != (SIZE_MARK ^ _p_mblock->size_)) {
00061:         return false;
00062:     }
00063:     return true;
00064: }
```

图 22.15

mblock_integrate()函数用于对 mblock_t 数据结构中的 next_not_和 size_not_两个变量进行设置。从其实现读者可以看出，它的计算并不是简单地取反，而是与特定的数值（分别由 ADDRESS_MARK 和 SIZE_MARK 变量表示）进行异或操作，特定数值可以任意定义。

mblock_integrity_check()函数的功用是用来校验 mblock_t 数据结构的完整性的，当校验不通过时返回 false，否则返回 true。

有了这两个函数后，对于所有需要更改 mblock_t 数据结构的地方，都得使用 mblock_integrate()以更新 next_not_和 size_not_两个变量。同样地，如果需要使用某一个 mblock_t 数据结构，那么在使用之前先要调用 mblock_integrity_check()函数以确保其数据是可信的。另外，当检查出数据结构已被破坏时，应当报错且终止后续的操作。

增加了管理数据完整性检查后，heap_init()、heap_alloc()和 heap_free()三个函数的实现都需要做相应的调整，在此不一一列出，读者可以自行查看。需要提醒的是，在当前的实现中一旦

mblock_integrity_check()函数发现了数据完整性问题，它只是返回一个错误值，在实际的系统中完全可以输出错误日志并让整个系统停下来或重新启动。一旦发现了 mblock_t 数据结构被更改了，还希望程序继续可靠地提供服务吗？

22.1.3.2　简化内存释放函数

下面看一看如何去除内存释放函数 heap_free()的第二个参数。很容易想到的是，将每一块分配出来的内存块的大小记录在每一块被分配出来的内存块中，即在进行内存分配时，多分配一些空间用于记录被分配出来的内存块的大小。当用户释放内存时，就不需要提供每一块内存块的大小了。

为此，需要增加一个新的数据结构，用于记录分配出来的内存块的大小，这个数据结构就是 mhead_t，其定义如图 22.16 所示。

```
00046: typedef struct {
00047:     msize_t size_; // length of block in g_alignment_bytes unit
00048:     msize_t size_not_;
00049: } mhead_t;
00050:
00051: #define ptr2buf(_ptr) (((char*)_ptr) + (g_mhead_size << g_alignment_in_bits))
00052: #define buf2ptr(_ptr) (((char*)_ptr) - (g_mhead_size << g_alignment_in_bits))
00053: #define ptr2mhead(_ptr) (((mhead_t*)ptr2buf(_ptr)) - 1)
```

图 22.16

图中除了示例说明如 mhead_t 结构的定义外，还示例说明了与之相关的三个宏的定义，分别是 ptr2buf()、buf2ptr()和 ptr2mhead()。

mhead_t 中的 size_用于表示被分配出来的内存块的大小，其单位是 g_alignmnet_bytes。而 size_not_变量的作用与前面 mblock_t 数据结构中的 size_not_是一样的，用做数据的完整性检查。另外，还增加了 mhead_integrate()和 mhead_integrity_check()两个函数用于对 mhead_t 结构进行完整性处理，这与前面讲到的对于 mblock_t 数据结构的完整性处理是类似的，这两个函数的实现如图 22.17 所示。

```
00066: static inline void mhead_integrate (register mhead_t* _p_mhead)
00067: {
00068:     _p_mhead->size_not_ = SIZE_MARK ^ _p_mhead->size_;
00069: }
00070:
00071: static inline bool mhead_integrity_check (register const mhead_t* _p_mhead)
00072: {
00073:     if (_p_mhead->size_not_ != (SIZE_MARK ^ _p_mhead->size_)) {
00074:         return false;
00075:     }
00076:     return true;
00077: }
```

图 22.17

图 22.18 示例说明了在一块被分配出来的内存块中，mhead_t 数据结构与用户数据区的布局。

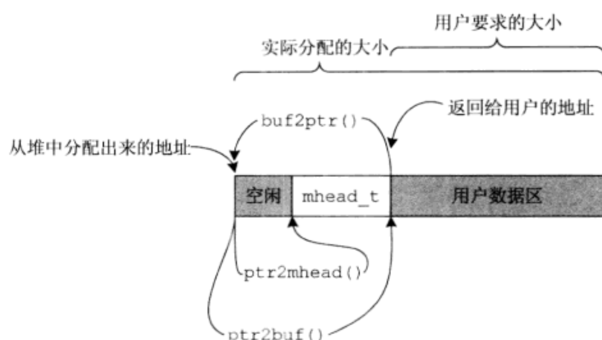


图 22.18

需要注意的是，图中的那块空闲区有可能不存在，这与内存块的边界对齐字节数有关。对于一个 32 位处理器，mhead_t 数据结构占用的字节数应当是 8 字节，如果此时设置的内存分配对齐数是 16 字节，在这种情况下，需要多分配的内存是 16 字节，图中的空闲区将为 8 字节。还有，mhead_t 数据结构总是放在与用户数据区挨着的位置，而不是放在开始处，为什么这样做后面会讲到。

图 22.18 还示例说明了增加的几个宏的作用。ptr2buf()宏是通过从堆中分配出来的起始地址返回最终给用户使用的内存首地址。buf2ptr()宏的作用则相反。ptr2mhead()宏返回的是 mhead_t 数据结构的开始位置。这几个宏在分配和释放内存时都需要使用到。

在增加了 mblock_t 数据结构完整性的保护，以及保存所分配内存的大小后，heap_alloc()函数的代码如图 22.19 所示。

```
00122: void* heap_alloc (msize_t _size)
00123: {
00124:     register mblock_t *p_pre, *p_next, *p_superfluous;
00125:     msize_t size_required, size_superfluous;
00126:     interrupt_level_t level;
00127:     mhead_t *p_mhead;
00128:
00129:     if (!g_initialized || is_in_interrupt () || 0 == _size) {
00130:         return null;
00131:     }
00132:
00133:     // convert the _size into g_alignment_bytes unit
00134:     size_required = ((_size + g_alignment_bytes - 1) >> g_alignment_in_bits);
00135:     size_required += g_mhead_size;
00136:     if (g_min_alloc_size > size_required) {
00137:         size_required = g_min_alloc_size;
00138:     }
00139:     level = global_interrupt_disable ();
00140:     // check whether have enough memory for this allocation request to fast failure
```

```
00141:     if (!mblock_integrity_check (&g_mblock_free) ||
00142:         (g_mblock_free.size_ < size_required)) {
00143:         global_interrupt_enable (level);
00144:         return null;
00145:     }
00146:     p_pre = p_next = &g_mblock_free;
00147:     for (;;) {
00148:         p_next = (mblock_t *)p_next->next_;
00149:         if (0 == p_next || !mblock_integrity_check (p_next)) {
00150:             // till now we have traversed all the block and didn't find
00151:             // a block for this request
00152:             global_interrupt_enable (level);
00153:             return null;
00154:         }
00155:         // if this block size cannot meet the request, continue next block
00156:         if (p_next->size_ < size_required) {
00157:             p_pre = p_next;
00158:             continue;
00159:         }
00160:         // block size is bigger or equal to size_required, get the
00161:         // remaining free size
00162:         size_superfluous = p_next->size_ - size_required;
00163:         if (size_superfluous <= g_min_alloc_size) {
00164:             size_required = p_next->size_;
00165:             p_superfluous = (mblock_t *)p_next->next_;
00166:             break;
00167:         }
00168:         // put the superfluous block into the free list
00169:         p_superfluous = (mblock_t *)((address_t)p_next +
00170:             (size_required << g_alignment_in_bits));
00171:         p_superfluous->next_ = p_next->next_;
00172:         p_superfluous->size_ = size_superfluous;
00173:         mblock_integrate (p_superfluous);
00174:         break;
00175:     }
00176:     p_pre->next_ = (address_t)p_superfluous;
00177:     mblock_integrate (p_pre);
00178:     g_mblock_free.size_ -= size_required;
00179:     mblock_integrate (&g_mblock_free);
00180:     global_interrupt_enable (level);
00181:     //lint -e{826}
00182:     p_mhead = ptr2mhead (p_next);
00183:     p_mhead->size_ = size_required;
00184:     mhead_integrate (p_mhead);
00185:     return ptr2buf (p_next);
00186: }
```

图 22.19

其中最大的变化是增加了几处对mblock_integrity_check()和mblock_integrate()函数的调用。当我们需要设置 mblock_t 数据结构时，需要调用 mblock_integrate()函数以更新用于校验变量中的值，比如第 173、177 和 179 行。另外，在使用每一个 mblock_t 数据结构之前，得先调用 mblock_integrity_check()去检查它的完整性，如第 141 和 149 行。第 182～183 行用于生成记录内存块大小的 mhead_t 数据结构，第 184 行 mhead_integrate()函数的调用是为了生成完整性校验数据。

heap_init()和 heap_free()函数也有类似的更改，在此不再列出。读者可以通过文件比较器[3]去了解 v1 与 v2 版本程序实现的具体差异是什么。

22.1.4 支持内存泄漏检测

内存泄漏是指不再需要的动态内存没有调用释放函数将其还给内存管理模块。内存泄漏的最终结果是耗尽所有的内存。在做进一步改进之前，我们需要先了解在嵌入式应用开发时通常有哪些手段用于防范和检测内存泄漏。

方法一是采用代码审查进行控制。这种方法是最容易想到的，但是效果也是相当的有限。当程序的复杂度增加时，这种方法就越加显得无效了。

方法二是通过使用一定的工具来帮助发现内存泄漏。比如来自 IBM 的 Purify、开源的 Valgrind 等。这些工具在使用时都不需要我们去改变程序源代码，可分为两类。一类是需要对代码与工具库进行重新编译。这类工具使用起来相对麻烦，通常需要将工具与项目的编译环境进行整合以便使用起来更加方便。Purify 工具就属于这类工具。另一类则不需要对代码进行重新编译，因此使用起来比较方便。Valgrind 工具位于其列。

使用工具检测内存泄漏问题应当注意两点。第一，要保证代码在测试时有尽可能高的代码覆盖率（参见第 29 章）。这是因为内存泄漏检测工具依赖于被检测的代码是否被运行到了，也只有被测代码被运行过了检测工具才能发现其中的内存泄漏问题。然而，我们往往很难做到百分之百的代码覆盖率，因此检测的效果也是有限的。第二，这些工具对于被测代码的性能有极大的影响[4]。如果检测工具的使用造成程序无法达到性能要求时，这类工具就无法使用。以上两点说明通过工具进行内存泄漏检测不可能成为终级的解决方法。另外，这些工具大都不能直接运行于资源有限的嵌入式系统中，这也限制了工具的运用环境。

方法三是采用一定的封装技术对内存的分配与释放进行接管。比如，提供一个模块对 C 库中的 malloc()和 free()函数进行很薄的封装，然后向应用程序提供相应的接口函数用于分配和释放内存。除此之外，封装层还提供一定的方式让我们能实时地得到运行时内存的使用情况。例如，可以看一看刚过去的 30 分钟内有哪些模块分配了内存且还没有释放。

上面三种方法，前两种属于非运行时的，而第三种则属于运行时的。运行时的方法有很大的一个优势，即产品即使是部署在现场，也可以在需要时通过一定的方法查看系统中是否存在内存泄漏问题。现实中，产品在现场的运行环境比实验室的要复杂，也就是说，我们不可能在实验室穷尽所有的测试用例，而采用运行时的方法将有助于发现在实验室中无法发现的内存泄漏问题。

---

③ 开源的 WinMerge 是一款不错的文件比较器。Araix Merge 是作者所使用过的最好的文件比较器，但它是商用软件。

④ 这是因为测试工具需要对每一块内存的分配与释放进行记录和分析。

从解决方案来看，尽管第三种方法最好，但也存在一些值得我们关注的问题。其一，由于内存管理模块对于每一次内存分配需要记录它是在程序的哪一处发生的，以便在需要时显示这些信息帮助定位内存泄漏点，因此存在额外的内存开销。其二，由于增加了一层封装，尽管很薄但内存的分配速度还是会有一点点下降，下降程度取决于管理算法。从以上两点来看，第三种方法是通过时间和空间来换取实用性的。

值得一提的是，方法二中的工具除了具备内存泄漏检测功能外，还能用于检测其他的问题。比如，用于检查是否存在使用没有初始化的指针、内存溢出等问题，而这些功能在方法三中是做不到的，或者说即使做到所带来的运行时开销也很大。因此，在现实项目中通常会将多种方法结合使用。

为了发现内存泄漏问题，需要掌握每次内存的分配情况。最为常见的做法是，每次分配都记录它所发生的文件名及在文件中的行号。这需要用到 C 语言中的两个宏——__FILE__和__LINE__。如果要使用这一方法来改进 mblock 算法以使其具备检测内存泄漏的功能，那么需要对 mhead_t 数据结构进行更改，即在数据结构中增加两个变量分别用于记录内存分配发生时的文件名和文件行号，图 22.20 列出了可能的改变。

```
typedef struct {
    msize_t size_;
    msize_t size_not_;
    const char *file_name_;
    int line_;
} mhead_t;
```

图 22.20

其中的 file_name_变量用于保存文件名，而 line_变量用于保存行号。除了数据结构的改变外，还得对 heap_alloc()函数的输入参数进行更改，即增加传递文件名和行信息的参数。最后，调用函数的地方也得使用__FILE__和__LINE__两个宏作为参数，如图 22.21 所示。

```
error_t mem_alloc (msize_t _size, const char *_file_name, int _line);
```

```
p_buf1 = heap_alloc (1, __FILE__, __LINE__);
```

图 22.21

对于这种常见做法，我们需要分析一下它所带来的资源开销。所带来的开销除了 mhead_t 数据结构中增加的 file_name_和 line_两个变量所占用的内存外，还有一部分内存是很容易被忽略的，即__FILE__宏所占用的内存。从 C 语言的角度来看，__FILE__是一个字符串，为源程序的文件名。比如，图 22.20 中__FILE__宏最终指向字符串“main.c”，这占用 7 个字节（包括字符结束符'\0'）。显然，如果文件名长的话所付出的内存开销将更大。注意，如果一个文件中

使用了多次__FILE__，内存不会成比例增长，对于同一个文件中的所有__FILE__宏，它们将共享同一个字符串。

除了资源开销较大外，这种常用方法在显示内存分配情况时还存在信息不是很直观的问题，即同一个位置多次分配的内存块打印出来的是多条离散的记录。虽然我们可以考虑在打印信息之前进行一定的同类项合并，但这需要花费一定的处理器时间。简单说来，当需要查看内存分配情况时，我们希望看到的信息是指示某一文件的某一行没有释放的次数。

如果要改善上面所谈到的常用方法，我们需要思考“对于哪一文件的哪一行，我们一定要用文件名加上一个行号来表示吗？能不能只用一个数字呢？”下面我们将探索另一种不同的方案。

先看一看图 22.22，其中引入了一个新的数据结构 mlocation_t，用来定义一些数字，每个数字将用于表示某个文件的某一内存分配行，后面称这个数字为位置标识。图中定义了从 EXAMPLE_MAIN_1 到 EXAMPLE_MAIN_4 四个位置标识。当需要增加位置标识时，只需将相应的位置标识名放入 LOCATIONS 宏中即可。

```
00029: #define LOCATIONS \
00030:     LOCATION(EXAMPLE_MAIN_1) \
00031:     LOCATION(EXAMPLE_MAIN_2) \
00032:     LOCATION(EXAMPLE_MAIN_3) \
00033:     LOCATION(EXAMPLE_MAIN_4) \
00034:
00035: #define LOCATION(a) a,
00036: typedef enum {
00037:     LOCATIONS
00038:     MLOCATION_END
00039: } mlocation_t;
00040: #undef LOCATION
```

图 22.22

是不是对于每一处调用 heap_alloc()函数的地方，都得在 mlocation_t 中增加一个新的定义值呢？不一定！

检测内存泄漏的重点是需要得到各内存的分配位置信息，分配信息可以细到对每一个分配点进行统计，但也可以粗到从模块级进行统计。当统计粒度大时，同一模块可以共用一个位置标识。采用粗粒度还是细粒度需要考虑两点。第一点是内存开销，第二点是项目开发时的方便性和程序的开发效率[5]。很显然，如果希望获得的统计信息精确，则每一处分配内存的地方应使用不同的位置标识，那么一旦发现内存泄漏就能明确地知道是哪一处。反之，若统计信息粒度较粗，当发现内存泄漏时，只能知道可能的几处，为了明确知道哪一处，可能还得对程序做

⑤ 如果所有的位置标识都放在一个文件中进行定义，则一旦对这个文件进行更改，所有依赖这个文件的源文件在下一次项目编译时都得重新编译，这有可能影响项目的编译效率，进而影响开发速度。

进一步的分析或重新编译程序并测试。无论如何，位置标识的使用具有一定的灵活性，我们完全可以根据应用情形进行一定的平衡。当然，最好的方法是将位置标识像第 15 章的错误码那样分模块进行管理，这一改进工作交由读者去完成。

除了定义各位置标识外，还得增加一个 loc2str()函数。该函数将返回某一位置标识的字符名称以便显示时使用。loc2str()函数的实现如图 22.23 所示。

embedded/code/platform/memory/heap/v3/inc/mlocation.c

```
00028: #define LOCATION(a) #a,
00029: static const char *g_locations [] = {LOCATIONS};
00030: #undef LOCATION
00031:
00032: const char *loc2str (mlocation_t _loc)
00033: {
00034:     if (_loc >= MLOCATION_END) {
00035:         return "invalid location number";
00036:     }
00037:     return g_locations [_loc];
00038: }
```

图 22.23

由于所有的位置标识是按顺序从 0 由低到高排好序的，因此它很适合作为数组的下标，也就是说，可以用数组来记录每一个位置标识的分配次数。数组的定义如图 22.24 所示，其中的 g_mlocation_count 数组就是用来存放每一个位置标识所对应的内存分配次数的。

embedded/code/platform/memory/heap/v3/[illegible]

```
00050: // reference for each memory allocation location
00051: static msize_t g_mlocation_count [MLOCATION_END];
```

图 22.24

位置标识需要在用户申请内存时作为一个参数传递给内存管理模块，也就是说，我们得为 heap_alloc()函数增加一个参数。那 heap_free()函数呢？显然，我们并不希望在释放内存时也要求用户提供一个位置标识，解决方法就是借鉴前面关于释放内存时不需要用户提供内存大小的方法，即当 heap_alloc()分配内存时将位置标识记录到被分配出去的内存块中，这可以采用前面引入 mhead_t 数据结构记录内存块大小的方法，只不过这一次是定义一个新的数据结构——mtail_t，如图 22.25 所示。

[illegible]

```
00052: typedef struct {
00053:     maddr_t loc_;
00054:     maddr_t loc_not_;
00055: } mtail_t;
00056:
00060: #define ptr2mtail(_ptr, _size) (mtail_t *)(((char*)_ptr) + \
00061:     ((_size - g_mtail_size) << g_alignment_in_bits))
00062:
00069: void* heap_alloc (msize_t _size, mlocation_t _loc, error_t *_p_error);
```

图 22.25

mtail_t 数据结构中的 loc_变量的类型虽然是 maddr_t，但实际上记录的是每一块内存分配的位置标识。同样同时存在 loc_not_变量用于完整性校验。另外，ptr2mtail()宏用于获取 mtail_t 结构所在的位置，图 22.26 示例说明了这个宏的作用。从图中可以看出，用户数据区是被 mhead_t 和 mtail_t 紧紧地夹着的，在后面我们会看到这一实现所带来的好处。再提醒一下，图中的两块空闲区有可能因为字节对齐数而不存在。

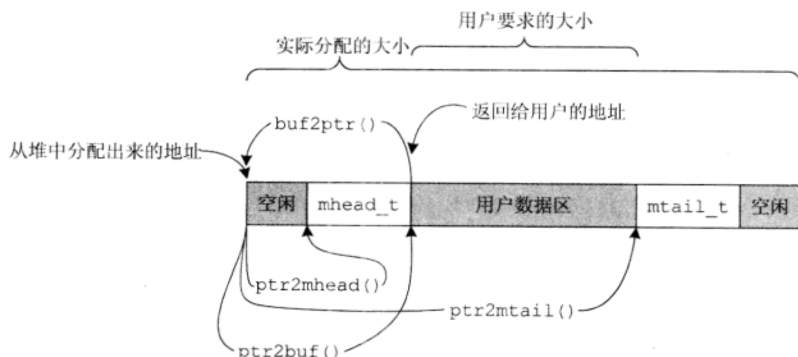


图 22.26

从图 22.25 中 heap_alloc()函数的声明还可以看出，除了增加_loc 参数用于指示内存分配时的位置标识外，还增加了一个_p_error 参数用于返回错误码。

图 22.27 列出了增加 mtail_t 结构后所增加的两个用于完整性处理的函数，即 mtail_integrate()和 mtail_integrity_check()函数，它们的功能不再赘述。

```
00083: static inline void mtail_integrate (register mtail_t* _p_mtail)
00084: {
00085:     _p_mtail->loc_not_ = LOC_MARK ^ _p_mtail->loc_;
00086: }
00087:
00088: static inline bool mtail_integrity_check (register const mtail_t* _p_mtail)
00089: {
00090:     if (_p_mtail->loc_not_ != (LOC_MARK ^ _p_mtail->loc_)) {
00091:         return false;
00092:     }
00093:     return true;
00094: }
```

图 22.27

增加了 mtail_t 结构后的 heap_alloc()函数实现如图 22.28 所示。

```
00137: void* heap_alloc (msize_t _size, mlocation_t _loc, error_t *_p_error)
00138: {
00139:     register mblock_t *p_pre, *p_next, *p_superfluous;
00140:     msize_t size_required, size_superfluous;
00141:     interrupt_level_t level;
```

```
00142:     mhead_t *p_mhead;
00143:     mtail_t *p_mtail;
00144:
00145:     if (!g_initialized) {
00146:         *_p_error = ERROR_T (ERROR_HEAP_ALLOC_NOTINIT);
00147:         return null;
00148:     }
00149:     if (is_in_interrupt ()) {
00150:         *_p_error = ERROR_T (ERROR_HEAP_ALLOC_INVCONTEXT);
00151:         return null;
00152:     }
00153:     if (0 == _size) {
00154:         *_p_error = ERROR_T (ERROR_HEAP_ALLOC_INVSIZE);
00155:         return null;
00156:     }
00157:     if (_loc >= MLOCATION_END) {
00158:         *_p_error = ERROR_T (ERROR_HEAP_ALLOC_INVLOC);
00159:         return null;
00160:     }
00161:
00162:     // convert the _size into g_alignment_bytes unit
00163:     size_required = ((_size + g_alignment_bytes - 1) >> g_alignment_in_bits);
00164:     size_required += g_mhead_size + g_mtail_size;
00165:     level = global_interrupt_disable ();
00166:     // check whether have enough memory for this allocation request to fast failure
00167:     if (!mblock_integrity_check (&g_mblock_free) ||
00168:         (g_mblock_free.size_ < size_required)) {
00169:         global_interrupt_enable (level);
00170:         *_p_error = ERROR_T (ERROR_HEAP_ALLOC_NOMEM1);
00171:         return null;
00172:     }
00173:     p_pre = p_next = &g_mblock_free;
00174:     for (;;) {
00175:         p_next = (mblock_t *)p_next->next_;
00176:         if (0 == p_next || !mblock_integrity_check (p_next)) {
00177:             // till now we have traversed all the block and didn't find
00178:             // a block for this request
00179:             global_interrupt_enable (level);
00180:             *_p_error = ERROR_T (ERROR_HEAP_ALLOC_NOMEM2);
00181:             return null;
00182:         }
00183:         // if this block size cannot meet the request, continue next block
00184:         if (p_next->size_ < size_required) {
00185:             p_pre = p_next;
00186:             continue;
00187:         }
00188:         // block size is bigger or equal to size_required, get the
00189:         // remaining free size
00190:         size_superfluous = p_next->size_ - size_required;
00191:         if (size_superfluous <= (g_mhead_size + g_mtail_size)) {
00192:             size_required = p_next->size_;
00193:             p_superfluous = (mblock_t *)p_next->next_;
00194:             break;
00195:         }
00196:         // put the superfluous block into the free list
00197:         p_superfluous = (mblock_t *)((address_t)p_next +
00198:             (size_required << g_alignment_in_bits));
00199:         p_superfluous->next_ = p_next->next_;
00200:         p_superfluous->size_ = size_superfluous;
00201:         mblock_integrate (p_superfluous);
```

```
00202:          break;
00203:      }
00204:      p_pre->next_ = (address_t)p_superfluous;
00205:      mblock_integrate (p_pre);
00206:      g_mblock_free.size_ -= size_required;
00207:      mblock_integrate (&g_mblock_free);
00208:      g_mlocation_count [_loc] ++;
00209:      global_interrupt_enable (level);
00210:      //lint -e{826}
00211:      p_mhead = ptr2mhead (p_next);
00212:      p_mhead->size_ = size_required;
00213:      mhead_integrate (p_mhead);
00214:      //lint -e{826}
00215:      p_mtail = ptr2mtail (p_next, size_required);
00216:      p_mtail->loc_ = (address_t)_loc;
00217:      mtail_integrate (p_mtail);
00218:      *_p_error = 0;
00219:      return ptr2buf (p_next);
00220: }
```

图 22.28

图中增加了多行设置_p_error 参数，以返回内存分配失败时的错误码，比如其中的第 170 和 180 行都返回一个错误码，而第 218 行返回 0 表示分配成功。错误码只有当 heap_alloc()函数返回 null 时才需要关注。

第 208 行对 g_mlocation_count 数据中位置标识所对应的元素进行加 1 操作，表示参数_loc 所代表的位置又进行了一次内存分配。第 215～217 行在分配出去的内存块的后面创建一个 mtail_t 数据结构，并对其中的 loc_变量使用_loc 参数进行初始化。

更新后 heap_free()函数的实现如图 22.29 所示。第 249 行新增了 p_mtail 局部变量。第 248～251 行对被释放内存的 mtail_t 数据结构进行校验。第 265 和 290 行分别就不同的情形对分配次数进行减 1 操作，表示内存又释放了一次。注意，位置标识是从内存块的 mtail_t 数据结构中获取到的。

```
00222: error_t heap_free (const void* _p_buf)
00223: {
00224:      register mblock_t* p_pre,*p_next;
00225:      interrupt_level_t level;
00226:      //lint -e{826}
00227:      register address_t p_ptr = (address_t) buf2ptr (_p_buf);
00228:      //lint -e{826}
00229:      mhead_t* p_mhead = ptr2mhead(p_ptr);
00230:      mtail_t* p_mtail;
00231:
00232:      if (!g_initialized) {
00233:          return ERROR_T (ERROR_HEAP_FREE_NOTINIT);
00234:      }
00235:      if (is_in_interrupt () && STATE_UP == system_state ()) {
00236:          return ERROR_T (ERROR_HEAP_ALLOC_INVCONTEXT);
00237:      }
00238:      if (0 == p_ptr || ((p_ptr & (g_alignment_bytes - 1)) != 0) ||
00239:          //lint -e{774}
```

```
00240:          (p_ptr < g_heap_addr_start) || (p_ptr > g_heap_addr_end)) {
00241:          return ERROR_T (ERROR_HEAP_FREE_INVBUF);
00242:      }
00243:      if (!mhead_integrity_check (p_mhead)) {
00244:          return ERROR_T (ERROR_HEAP_FREE_INVMHEAD);
00245:      }
00246:
00247:      //lint -e{826}
00248:      p_mtail = ptr2mtail (p_ptr, p_mhead->size_);
00249:      if (!mtail_integrity_check (p_mtail)) {
00250:          return ERROR_T (ERROR_HEAP_FREE_INVMTAIL);
00251:      }
00252:      level = global_interrupt_disable ();
00253:      if (!mblock_integrity_check (&g_mblock_free)) {
00254:          global_interrupt_enable (level);
00255:          return ERROR_T (ERROR_HEAP_FREE_INVMBLOCK1);
00256:      }
00257:      g_mblock_free.size_ += p_mhead->size_;
00258:      mblock_integrate (&g_mblock_free);
00259:      if (0 == g_mblock_free.next_) {
00260:          g_mblock_free.next_ = p_ptr;
00261:          p_next = (mblock_t *)p_ptr;
00262:          p_next->size_ = p_mhead->size_;
00263:          p_next->next_ = 0;
00264:          mblock_integrate (p_next);
00265:          g_mlocation_count [p_mtail->loc_] --;
00266:          global_interrupt_enable (level);
00267:          return 0;
00268:      }
00269:      else {
00270:          p_pre = &g_mblock_free;
00271:          p_next = (mblock_t *)g_mblock_free.next_;
00272:      }
00273:      if (!mblock_integrity_check (p_pre) ||
00274:          !mblock_integrity_check (p_next)) {
00275:          global_interrupt_enable (level);
00276:          return ERROR_T (ERROR_HEAP_FREE_INVMBLOCK2);
00277:      }
00278:      // find the right position of the list
00279:      while (p_pre->next_ < p_ptr) {
00280:          p_pre = p_next;
00281:          p_next = (mblock_t *)p_next->next_;
00282:          if (null == p_next) {
00283:              break;
00284:          }
00285:          if (!mblock_integrity_check (p_next)) {
00286:              global_interrupt_enable (level);
00287:              return ERROR_T (ERROR_HEAP_FREE_INVMBLOCK3);
00288:          }
00289:      }
00290:      g_mlocation_count [p_mtail->loc_] --;
00291:      // merge with the previously adjacent block if needed
00292:      if ((((p_pre->size_ << g_alignment_in_bits) + (address_t)p_pre)
00293:          == p_ptr) && (p_pre != &g_mblock_free)) {
00294:          p_pre->size_ += p_mhead->size_;
00295:          mblock_integrate (p_pre);
00296:      }
00297:      else {
00298:          p_pre->next_ = p_ptr;
00299:          mblock_integrate (p_pre);
```

```
00300:         p_pre = (mblock_t *)p_ptr;
00301:         p_pre->size_ = p_mhead->size_;
00302:         p_pre->next_ = (address_t)p_next;
00303:         mblock_integrate (p_pre);
00304:     }
00305:     if (0 == p_next) {
00306:         // this is the last block no more mergence is needed
00307:         global_interrupt_enable (level);
00308:         return 0;
00309:     }
00310:     // merge with the following adjacent block if needed
00311:     if (((p_pre->size_ << g_alignment_in_bits) + (address_t)p_pre)
00312:         == (address_t)p_next) {
00313:         p_pre->size_ += p_next->size_;
00314:         p_pre->next_ = p_next->next_;
00315:         mblock_integrate (p_pre);
00316:     }
00317:     global_interrupt_enable (level);
00318:     return 0;
00319: }
```

图 22.29

heap_dump()函数也需要做相应的更改，以包含输出各位置标识所进行的内存分配次数，在此并没有将更改的源代码列出，请读者自行查看。

最后一处更改位于 module_heap()函数内。当系统进行终止化时，需要检查 g_mlocation_count 数组中的各元素是否为 0，不为 0 则说明存在内存泄漏问题。出现这种情形时，通过日志显示出来。更改后的 module_heap()函数的代码如图 22.30 所示，其中增加的第 396～402 行正是对 g_mlocation_count 进行检查，并在需要时输出错误日志。

```
00385: error_t module_heap (system_state_t _state)
00386: {
00387:     if (STATE_INITIALIZING == _state) {
00388:         heap_info_t heap_info;
00389:         heap_info_get (&heap_info);
00390:         return heap_init (heap_info.start_, heap_info.end_,
00391:             heap_info.alignment_in_bits_);
00392:     }
00393:     else if (STATE_DESTROYING == _state) {
00394:        msize_t size = ((g_heap_size - g_mblock_free.size_) << g_alignment_in_bits);
00395:         if (0 != size) {
00396:             for (mlocation_t loc = (mlocation_t)0; loc < MLOCATION_END; ++ loc) {
00397:                 if (0 == g_mlocation_count [loc]) {
00398:                     continue;
00399:                 }
00400:                 console_print ("Error: memory leak point %s (%d)\n",
00401:                     loc2str (loc), g_mlocation_count [loc]);
00402:             }
00403:         }
00404:     }
00405:     return 0;
00406: }
```

图 22.30

图 22.31 是 heapv3 示例程序的运行结果。heapv3 示例程序的源代码与 heapv1 示例程序的源代码几乎相同，其中唯一的一处变化是注释了第 8 个测试用例以模拟内存泄漏的情形。从 heapv3 示例程序的输出结果可以看出，每一次调用 heap_dump()函数时都将输出所有发生内存分配的位置标识和其分配次数。很显然，采用位置标识的形式有助于节约用于管理的内存，且最后报告的信息不是“流水账”。

```
                          /embedded/build
$ make
                          /embedded/build
$ ./release/heapv3.exe

System is going to be up

……显示结果有删减……

=========================
Test Case 6): free 1 byte ->
=========================
   Freed Addr: 0x7e6e0010

Free Block(s)
-------------
  Block Start Addr: 7e6e0008, size: 8028 (32808)
  Block Start Addr: 7e708050, size: fd7fb8 (16613304)

Allocation Count
----------------
  EXAMPLE MAIN 3: 1
  EXAMPLE MAIN 4: 1

Summary
-------
  Alignment: 8
  Heap Addr: 7e6e0008
      Total: 1000000 (16777216)
       Used: 20020 (131104)
       Free: fdffc0 (16646112)

=========================
Test Case 7): free 64K bytes ->
=========================
   Freed Addr: 0x7e6f8048

Free Block(s)
-------------
  Block Start Addr: 7e6e0008, size: 8028 (32808)
  Block Start Addr: 7e6f8040, size: fe7fc8 (16678856)

Allocation Count
----------------
  EXAMPLE MAIN 3: 1

Summary
-------
  Alignment: 8
  Heap Addr: 7e6e0008
      Total: 1000000 (16777216)
       Used: 10010 (65552)
```

```
    Free: fefff0 (16711664)

System is going to be down
Error:  memory leak point EXAMPLE MAIN 3 (1)
```

图 22.31

从这个示例程序可以看出，程序在终止化时输出了一行报告内存泄漏的错误日志。

22.1.5　实现内存溢出检测

内存溢出是指使用内存时超出了用户数据区，图 22.32 示例说明了从堆中获得内存的可使用区。图 22.33 所示程序片段的第 130 和 132 行将造成内存溢出。

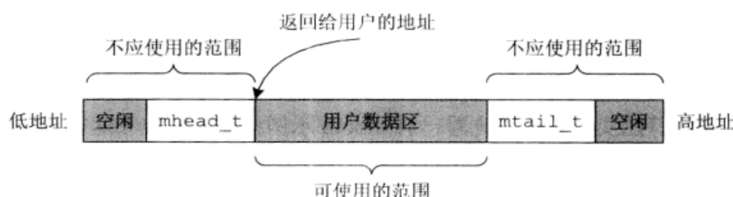


图 22.32

```
00123: char *p_char = (char *)heap_alloc (1, EXAMPLE_MAIN_1, &result);
00124: if (0 != result) {
00125:     printf ("mem_alloc () returns %s!\n", errstr (result));
00126:     return -1;
00127: }
00128:
00129: // overflow
00130: memset (p_char, 0, 18);
00131: // overflow
00132: p_char[-1] = 0x33;
```

图 22.33

前面提及，在设计时让 mhead_t 和 mtail_t 两块数据紧紧地夹住用户数据区，这样做的好处就是能用来检测（部分）内存写溢出问题。由于 mhead_t 和 mtail_t 结构都采用了完整性保护，当用户调用 heap_free()函数返还不再使用的内存时，heap_free()函数通过检查 mhead_t 和 mtail_t 数据结构的完整性就能发现写溢出问题。

当内存写溢出没有落入 mhead_t 和 mtail_t 两个数据结构所占用的内存区时，堆管理模块并不能发现内存写溢出。另外，如果出现了内存读溢出，堆管理模块更是无能为力。尽管这里的溢出检测功能很弱，但有总比没有好。

22.1.6　内存碎片问题

随着程序的运行，堆空间会因为内存的不断分配与释放而变得支离破碎。尽管存在内存合并

的功能，但仍有可能造成整个堆的空闲空间大于所请求的分配值而无法分配出内存这种情形。

从 mblock 算法来看，内存碎片的增加将使得内存分配与释放的速度发生变化。因为无论是分配还是释放，都需要遍历空闲内存块，而遍历的次数与分配和释放效率是正相关的。

很多堆管理算法会采用将堆分成各档固定内存块的方式以试图减小内存碎片。其思想是将内存块按 8、16、32、64 等 2 的 *n* 次方大小的形式进行分类，当向堆管理模块请求内存时，根据实际请求的大小，找到一个比它大且大小最接近的块进行分配。这种思想，其实就是部分采用内存池的思想来管理堆空间。

22.2 内存池管理

内存池管理算法的提出正是为了解决堆管理算法中的内存碎片问题。另外，由于它是以固定大小进行内存分配的，所以它具有更高的效率。接下来将要介绍的一种实现，作者称其为 mpool 算法。在介绍算法实现之前，先看一看 mpool 示例程序的实现和运行结果。

22.2.1 mpool 示例程序

图 22.34 是 mpool 示例程序的源代码。

```
00026: #include "main.h"
00027: #include "device.h"
00028: #include "mpool.h"
00029: #include "console.h"
00030:
00031: //lint -e754
00032:
00033: #define BUFFER_COUNT   32
00034:
00035: typedef struct {
00036:     char buf_ [32];
00037: } buffer_t;
00038:
00039:
00040: static void task_test (const char _name [], void *_p_arg)
00041: {
00042:     error_t error;
00043:     void *p_buf1, *p_buf2, *p_buf3, *p_buf4, *p_buf5;
00044:     MPOOL_MEMORY_DECLARE (pool_node, pool_buffer, buffer_t, BUFFER_COUNT);
00045:     mpool_handle_t handle;
00046:
00047:     UNUSED (_name);
00048:     UNUSED (_p_arg);
00049:
00050:     error = mpool_create ("Test", &handle, pool_node, pool_buffer,
00051:         sizeof (buffer_t), sizeof (pool_buffer)/sizeof (pool_buffer [0]));
00052:     if (0 != error) {
00053:         console_print ("Error: mpool_create() returns %s!\n", errstr (error));
00054:         multitasking_stop ();
```

```
00055:     }
00056:
00057:     console_print ("\n");
00058:     console_print ("===========\n");
00059:     console_print ("Before Test ->\n");
00060:     console_print ("===========\n");
00061:     mpool_dump ();
00062:
00063:     console_print ("================================\n");
00064:     console_print ("Test Case 1): allocate a buffer ->\n");
00065:     console_print ("================================\n");
00066:     p_buf1 = mpool_buffer_alloc (handle);
00067:     console_print (" Allocated Addr: %p\n\n", p_buf1);
00068:     mpool_dump ();
00069:
00070:     console_print ("======================================\n");
00071:     console_print ("Test Case 2): allocate 4 more buffers ->\n");
00072:     console_print ("======================================\n");
00073:     p_buf2 = mpool_buffer_alloc (handle);
00074:     p_buf3 = mpool_buffer_alloc (handle);
00075:     p_buf4 = mpool_buffer_alloc (handle);
00076:     p_buf5 = mpool_buffer_alloc (handle);
00077:     console_print (" Allocated Addr: %p\n", p_buf2);
00078:     console_print (" Allocated Addr: %p\n", p_buf3);
00079:     console_print (" Allocated Addr: %p\n", p_buf4);
00080:     console_print (" Allocated Addr: %p\n\n", p_buf5);
00081:     mpool_dump ();
00082:
00083:     console_print ("===============================\n");
00084:     console_print ("Test Case 3): free all buffers ->\n");
00085:     console_print ("===============================\n");
00086:     (void) mpool_buffer_free (handle, p_buf1);
00087:     (void) mpool_buffer_free (handle, p_buf2);
00088:     (void) mpool_buffer_free (handle, p_buf3);
00089:     (void) mpool_buffer_free (handle, p_buf4);
00090:     (void) mpool_buffer_free (handle, p_buf5);
00091:     mpool_dump ();
00092:
00093:     (void) mpool_delete (handle);
00094:     multitasking_stop ();
00095: }
00096:
00097: error_t module_testapp (system_state_t _state)
00098: {
00099:     static task_handle_t handle;
00100:     STACK_DECLARE (stack, 1024);
00101:
00102:     if (STATE_INITIALIZING == _state) {
00103:         (void) task_create (&handle, "Test", 16, stack, sizeof (stack));
00104:         (void) task_start (handle, task_test, 0);
00105:     }
00106:     else if (STATE_DESTROYING == _state) {
00107:         (void) task_delete (handle);
00108:     }
00109:     return 0;
00110: }
00111:
00112: int module_registration_entry (int argc, char *argv [])
00113: {
00114:     UNUSED (argc);
```

```
00115:    UNUSED (argv);
00116:
00117:    (void) module_register ("Interrupt", MODULE_INTERRUPT, CPU_LEVEL,
              module_interrupt);
00118:    (void) module_register ("Device", MODULE_DEVICE, DRIVER_LEVEL, module_device);
00119:    (void) module_register ("Timer", MODULE_TIMER, OS_LEVEL, module_timer);
00120:    (void) module_register ("Task", MODULE_TASK, OS_LEVEL, module_task);
00121:    (void) module_register ("Mpool", MODULE_MPOOL, OS_LEVEL, module_mpool);
00122:    (void) module_register ("TestApp", MODULE_TESTAPP, APPLICATION_LEVEL3,
              module_testapp);
00123:    return 0;
00124: }
```

图 22.34

一个内存池将包含多个缓冲区（buffer），第 33 行定义了缓冲区的个数。第 35～37 行定义了一个缓冲区所对应的数据结构，或者说，buffer_t 数据结构将决定缓冲区的大小。

task_test()函数的实现是重点。第 44 行通过使用 MPOOL_MEMORY_DECLARE 宏为内存池获取所需内存。其中 pool_node 和 pool_buffer 是两个数组变量名，前者被用做内存池的内部管理数据，后者是缓冲区内存；第三个参数是每一个缓冲区所对应的数据结构；最后一个参数指示缓冲区的个数。第 50 行通过调用 mpool_create()函数分配一个内存池。mpool_create()函数的第一个参数是内存池的名称；第二个参数是将要返回的内存池句柄；第三个和第四个参数分别是内存池的管理内存和缓冲区内存；第五个参数用于标识每一个缓冲区的大小，在这里它应当是 buffer_t 数据结构所占用的内存数；最后一个参数指示内存池中有多少个缓冲区。

task_test()函数中的后续程序就是几个测试用例，各用例通过调用 mpool_buffer_alloc()函数和 mpool_buffer_free()函数从内存池中分配和释放缓冲区，以及在每个测试用例的最后通过调用 mpool_dump()函数查看模块信息。在所有测试完成之后，需要调用 mpool_delete()函数释放内存池（第 93 行）。示例程序的运行结果如图 22.35 所示。

```
$ make
$ ./release/mpool.exe

System is going to be up

Before Test ->

Summary
-------
  Supported: 2
  Allocated: 1
  .BSS Used: 172

Pool Details
------------
  Name: Test
    Buffer Size: [illegible]
      Capacity: [illegible]
     Available: [illegible]
```

```
     No Buffer: 0

================================
Test Case 1): allocate a buffer ->
================================
   Allocated Addr: 0x40d030

Summary
-------
  Supported: 2
  Allocated: 1
  .BSS Used: 172

Pool Details
------------
  Name: Test
    Buffer Size: 32
       Capacity: 32
      Available: 31
      No Buffer: 0

=========================================
Test Case 2): allocate 4 more buffers ->
=========================================
   Allocated Addr: 0x40d050
   Allocated Addr: 0x40d070
   Allocated Addr: 0x40d090
   Allocated Addr: 0x40d0b0

Summary
-------
  Supported: 2
  Allocated: 1
  .BSS Used: 172

Pool Details
------------
  Name: Test
    Buffer Size: 32
       Capacity: 32
      Available: 27
      No Buffer: 0

====================================
Test Case 3): free all buffers ->
====================================
Summary
-------
  Supported: 2
  Allocated: 1
  .BSS Used: 172

Pool Details
------------
  Name: Test
    Buffer Size: 32
       Capacity: 32
      Available: 32
      No Buffer: 0

System is going to be down
```

图 22.35

如果只将测试程序中的第 90 行注释掉，即对 p_buf5 所指向的缓冲区不做释放操作，编译后运行的结果如图 22.36 所示。其中报告了内存池“Test”在被删除时缓冲区没有完全回收这一错误，隐含的意思是可能存在内存泄漏。这个检测动作是在 mpool_delete()函数中进行的。

```
$ make

$ ./release/mpool.exe

System is going to be up

……显示结果有删减……
Error: buffer is not freed completely before deleting mpool "Test"

System is going to be down
```

图 22.36

如果只将测试程序中的第 93 行注释掉，运行程序将获得图 22.37 所示的结果。其中告知“Test”内存池没有被删除。

```
$ make

$ ./release/mpool.exe

……显示结果有删减……
System is going to be down
Error: memory pool "Test" isn't deleted
```

图 22.37

有了这些感性的认识后，让我们一同看一看 mpool 算法的具体实现。

22.2.2 程序实现

管理内存池所需的数据结构如图 22.38 所示。

```
00035: #define MPOOL_MEMORY_DECLARE(_node_name, _buf_name, _type, _count)\
00036:     static mpool_node_t _node_name [_count]; \
00037:     static _type _buf_name [_count];
00038:
00039: typedef struct {
00040:     dll_node_t node_;
00041:     address_t addr_;
00042:     bool in_use_;
00043: } mpool_node_t;
00044:
00045: typedef struct {
00046:     dll_node_t node_;
00047:     magic_number_t magic_number_;
00048:     char name_ [NAME_MAX_LENGTH + 1];
00049:     address_t addr_start_;
00050:     address_t addr_end_;
00051:     msize_t buffer_size_;
```

```
00052:     msize_t buffer_count_;
00053:     int buffer_size_in_bits_;
00054:     bool apply_shift_;
00055:     dll_t free_buffer_;
00056:     mpool_node_t *p_node_;
00057:     // statistic
00058:     statistic_t stats_nobuf_;
00059: } mpool_t, *mpool_handle_t;
```

图 22.38

第 35 行所定义的 MPOOL_MEMORY_DECLARE()宏用于定义一个内存池所需的内存。一个内存池所需的内存被分成两大块，其中一块是缓冲区内存（给获取者使用），另一块是用于管理缓冲区的内存（模块内部使用）。从宏的实现来看，其就是通过定义静态数组的方式获得内存。

第 39～43 行定义了用于管理缓冲区的管理数据结构。一个缓冲区对应一个 mpool_node_t 类型的数据实例。node_表示链表节点；addr_变量用于记录被管理缓冲区的开始地址；in_used_变量用于表示所对应的缓冲区是否已被分配出去。

第 45～59 行定义了内存池所需的管理数据结构 mpool_t，以及其指针形式 mpool_ handle_t。mpool_t 各成员变量的作用如下：

（1）ClearRTOS 所支持的内存池都在 g_mpool_pool 数组（图 22.39 的第 39 行）中。一个内存池在没有被分配出来之前，将放入 g_free_mpool 链表中（图 22.39 的第 40 行），而一旦被分配出来就会放入 g_used_mpool 链表中（图 22.39 的第 41 行）。node_变量的作用就是当做链表节点。

（2）addr_start_和 end_addr_记录的是整个缓冲区的开始和结束地址。

（3）buffer_size_记录的是每个缓冲区所占内存字节数。

（4）buffer_count_记录的是一个内存池中有多少块缓冲区可供分配。

（5）当 buffer_size_的大小是 2 的 *n* 次方时，apply_shift_的值被设置为 true，否则为 false。当缓冲区的大小正好为 2 的 *n* 次方时，缓冲区释放操作可以采用移位的方式以提高程序的运行效率，apply_shift_变量的作用正是帮助标识是否可以采用移位操作。

（6）当缓冲区的大小是 2 的 *n* 次方时，buffer_size_in_bits_变量用于记录 *n* 的具体值，该值在需要进行移位操作时有用。

（7）free_buffer_是一个链表，用于存放空闲缓冲区块所对应的 mpool_node_t 结构，当用户需要分配缓冲区时，mpool_buffer_alloc()函数只需从 free_buffer_链表中取出一个就行了。采用这种方式，能以恒定的速度高效地完成缓冲区分配工作。

（8）p_node_用于保存管理内存区的开始地址，p_node_所指向的内存块正是通过

MPOOL_MEMORY_DECLARE 宏的第一个参数所获得的。

（9）stats_nobuf_用于统计内存池所出现的缓冲区不够用的次数。

```
00039: static mpool_t g_mpool_pool [CONFIG_MAX_MPOOL];
00040: static dll_t g_free_mpool;
00041: static dll_t g_used_mpool;
```

图 22.39

22.2.2.1 内存池创建

mpool_create()函数用于创建内存池，其实现如图 22.40 所示。

```
00043: static void mpool_init ()
00044: {
00045:     int idx;
00046:
00047:     dll_init (&g_free_mpool);
00048:     for (idx = 0; idx <= MPOOL_LAST_INDEX; idx ++) {
00049:         dll_push_tail (&g_free_mpool, &g_mpool_pool [idx].node_);
00050:     }
00051: }
00052:
00053: error_t mpool_create (const char _name [], mpool_handle_t *_p_handle,
00054:     void *_node, void *_buffer, msize_t _buffer_size, msize_t _buffer_count)
00055: {
00056:     static bool initialized = false;
00057:     mpool_node_t *p_node = _node;
00058:     address_t buffer_addr = (address_t) _buffer;
00059:     interrupt_level_t level;
00060:     mpool_handle_t handle;
00061:     usize_t index;
00062:
00063:     if (is_in_interrupt ()) {
00064:         return ERROR_T (ERROR_MPOOL_CREATE_INVCONTEXT);
00065:     }
00066:     if (null == _node || null == _buffer) {
00067:         return ERROR_T (ERROR_MPOOL_CREATE_INVPTR);
00068:     }
00069:
00070:     level = global_interrupt_disable ();
00071:     if (!initialized) {
00072:         mpool_init ();
00073:         initialized = true;
00074:     }
00075:     handle = (mpool_handle_t) dll_pop_head (&g_free_mpool);
00076:     if (null == handle) {
00077:         global_interrupt_enable (level);
00078:         return ERROR_T (ERROR_MPOOL_CREATE_NOPOOL);
00079:     }
00080:     dll_push_tail (&g_used_mpool, &handle->node_);
00081:     global_interrupt_enable (level);
00082:
00083:     dll_init (&handle->free_buffer_);
00084:     for (index = 0; index < _buffer_count; ++ index) {
```

```
00085:         p_node->addr_ = buffer_addr;
00086:         p_node->in_use_ = false;
00087:         dll_push_tail (&handle->free_buffer_, &p_node->node_);
00088:         p_node ++;
00089:         buffer_addr += _buffer_size;
00090:     }
00091:
00092:     if (0 == _name) {
00093:         handle->name_ [0] = 0;
00094:     }
00095:     else {
00096:         strncpy (handle->name_, _name, (usize_t)sizeof (handle->name_));
00097:         handle->name_ [sizeof (handle->name_) - 1] = 0;
00098:     }
00099:     handle->addr_start_ = (address_t) _buffer;
00100:     handle->addr_end_ = buffer_addr;
00101:     handle->p_node_ = _node;
00102:     handle->buffer_size_ = _buffer_size;
00103:     handle->buffer_count_ = _buffer_count;
00104:     handle->apply_shift_ = convert_to_shift_bits ((u32_t)
00105:         _buffer_size, &handle->buffer_size_in_bits_);
00106:     handle->magic_number_ = MAGIC_NUMBER_MPOOL;
00107:     *_p_handle = handle;
00108:     return 0;
00109: }
```

图 22.40

第 63～65 行检查函数是否是在中断状态下被调用的，如果是则返回错误。第 66～68 行检查内存池所需要的内存是否被提供。第 71～74 行首先查看模块是否已初始化，如果没有则调用 mpool_init()函数对其进行初始化。mpool_init()函数的实现位于第 43～51 行，其完成将 g_mpool_pool 数组中的各元素放入 g_free_mpool 链表中。第 75～80 行从空闲链表中获取一个内存池管理数据结构，并加入链表 g_used_mpool 中。

第 83 行初始化内存池管理数据结构中的 free_buffer_链。第 84～90 行将缓冲区通过对应的 mpool_node_t 数据结构放入该链表中以备后续分配。第 92～98 行对数据结构中的名称进行初始化。第 99～103 行记录缓冲区相关的信息。第 104 行通过调用 convert_to_shift_bits()函数将缓冲区的大小转换为移位比特数并放入 buffer_size_in_bits_变量中。当缓冲区的大小不是 2 的 n 次方时，convert_to_shift_bits()函数将返回 false，否则为 true。第 106 行设置标识以示有效。第 107 行将内存池返回给函数调用者。

22.2.2.2　内存池删除

删除一个内存池需要调用 mpool_delete()函数，其实现如图 22.41 所示。

```
00111: error_t mpool_delete (mpool_handle_t _handle)
00112: {
00113:     interrupt_level_t level;
00114:     bool all_freed;
00115:     char name [NAME_MAX_LENGTH + 1];
00116:
```

```
00117:     if (is_in_interrupt () && STATE_UP == system_state ()) {
00118:         return ERROR_T (ERROR_MPOOL_CREATE_INVCONTEXT);
00119:     }
00120:     level = global_interrupt_disable ();
00121:     if (is_invalid_handle (_handle)) {
00122:         global_interrupt_enable (level);
00123:         return ERROR_T (ERROR_MPOOL_DELETE_INVHANDLE);
00124:     }
00125:
00126:     _handle->magic_number_ = 0;
00127:     all_freed = (_handle->buffer_count_ == dll_size (&_handle->free_buffer_));
00128:     if (!all_freed) {
00129:         strncpy (name, _handle->name_, sizeof (name));
00130:     }
00131:     dll_remove (&g_used_mpool, &_handle->node_);
00132:     dll_push_tail (&g_free_mpool, &_handle->node_);
00133:     global_interrupt_enable (level);
00134:
00135:     if (!all_freed) {
00136:         console_print ("Error: buffer is not freed completely "
00137:             "before deleting mpool \"%s\"\n", name);
00138:     }
00139:     return 0;
00140: }
```

图 22.41

第 117 行的目的是防止函数在中断状态下被调用。第 121～124 行检查被释放的内存池是否是有效的。第 126 行将标识设置为 0，表示内存池不再有效。第 127 行检查内存池中的缓冲区是否都被回收了，后面需要根据这一信息输出错误日志。第 129 行将内存池的名称拷贝到局部变量中，以便后面打印时使用，显然这个拷贝动作只在输出日志时才需要。第 131 和 132 行将内存池从 g_used_mpool 链表中移除并放入 g_free_mpool 链表中。第 135～138 行检查被删除的内存池是否存在缓冲区没有回收现象，如果存在则通过错误日志加以提示。

22.2.2.3 分配缓冲区

图 22.42 是缓冲区分配函数 mpool_buffer_alloc()的实现。

```
00142: void* mpool_buffer_alloc (mpool_handle_t _handle)
00143: {
00144:     interrupt_level_t level;
00145:     mpool_node_t *p_node;
00146:
00147:     level = global_interrupt_disable ();
00148:     if (is_invalid_handle (_handle)) {
00149:         global_interrupt_enable (level);
00150:         return null;
00151:     }
00152:     p_node = (mpool_node_t *)dll_pop_head (&_handle->free_buffer_);
00153:     if (0 == p_node) {
00154:         _handle->stats_nobuf_ ++;
00155:         global_interrupt_enable (level);
00156:         return null;
00157:     }
```

```
00158:     global_interrupt_enable (level);
00159:     p_node->in_use_ = true;
00160:     return (void *)p_node->addr_;
00161: }
```

图 22.42

第 148 行先从内存池中获取一个管理空闲缓冲区的节点，前面说过了，一个管理节点对应于一个空闲缓冲区。第 153 行检查是否真正获得了一个管理节点，如果没有则说明缓冲区已被分配完，在第 154 行更新相应的统计信息，并在第 156 行返回 null 以示缓冲区分配失败。如果获得了一个有效的管理节点，则在第 159 行设置该节点已被分配。最后，在第 160 行将管理节点中所记录的缓冲区地址返回给函数调用者。注意，这个地址是在图 22.40 中 mpool_create() 函数的第 85 行初始化好的。

22.2.2.4　释放缓冲区

当一个缓冲区用完了后需要调用 mpool_buffer_free() 函数将其还回内存池，mpool_buffer_free()函数的实现如图 22.43 所示。

```
code/platform/memory/mpool/src/mpool.c
00163: error_t mpool_buffer_free (mpool_handle_t _handle, void* _p_buf)
00164: {
00165:     address_t free_addr = (address_t)_p_buf;
00166:     interrupt_level_t level;
00167:     msize_t index;
00168:
00169:     level = global_interrupt_disable ();
00170:     if (is_invalid_handle (_handle)) {
00171:         global_interrupt_enable (level);
00172:         return ERROR_T (ERROR_MPOOL_FREE_INVHANDLE);
00173:     }
00174:
00175:     if (free_addr < _handle->addr_start_ || free_addr >= _handle->addr_end_) {
00176:         global_interrupt_enable (level);
00177:         return ERROR_T (ERROR_MPOOL_FREE_OUTOFRANGE);
00178:     }
00179:     free_addr -= _handle->addr_start_;
00180:     if (_handle->apply_shift_ && 0 == (free_addr & (_handle->buffer_size_ - 1))){
00181:         index = free_addr >> _handle->buffer_size_in_bits_;
00182:     }
00183:     else if (0 == (free_addr % _handle->buffer_size_)) {
00184:         index = free_addr/_handle->buffer_size_;
00185:     }
00186:     else {
00187:         global_interrupt_enable (level);
00188:         return ERROR_T (ERROR_MPOOL_FREE_INVALIGNMENT);
00189:     }
00190:
00191:     if (_handle->p_node_[index].addr_ != (address_t)_p_buf) {
00192:         global_interrupt_enable (level);
00193:         return ERROR_T (ERROR_MPOOL_FREE_INVADDR);
00194:     }
00195:     if (_handle->p_node_[index].in_use_ != true) {
00196:         global_interrupt_enable (level);
00197:         return ERROR_T (ERROR_MPOOL_FREE_NOTINUSE);
```

```
00198:      }
00199:
00200:      _handle->p_node_[index].in_use_ = false;
00201:      dll_push_tail (&_handle->free_buffer_, &_handle->p_node_[index].node_);
00202:      global_interrupt_enable (level);
00203:
00204:      return 0;
00205: }
```

图 22.43

缓冲区在内存池中是一个挨着一个的，因为缓冲区内存的定义实际上是一个数组。通过缓冲区的地址，可以计算出它在数组中的索引号，并进而可以知道它所对应的管理节点，这是缓冲区释放函数的基本工作原理。

第 175～178 行检查所释放缓冲区的地址是否在内存池所管理的范围内。第 179 行将缓冲区的地址减去内存池所管理缓冲区的开始地址为后面计算数组索引做准备。如果缓冲区的大小是 2 的 n 次方，则在第 181 行通过移位的方式获得数组索引号。第 180 行除了检查 apply_shift_ 变量是否为 true 外，还得确保所释放的地址是满足一定的边界对齐要求的，这样做的目的是确保被释放的缓冲区地址是有效的。如果缓冲区的大小不是 2 的 n 次方，则数组索引号在第 184 行通过除法运算获得。同样地，做除法时在第 183 行通过求余的方式以检验被释放缓冲区地址的有效性。程序如果运行到了第 187 行，则意味着被释放的缓冲区地址是非法的，此时需返回相应的错误码。

第 191 行通过验证管理节点中记录的缓冲区地址是否与输入参数相一致，以做进一步的有效性检验。第 195 行对管理节点中的标志位进行检查，防止一个缓冲区被多次释放。这两步看似多余，但都是为了保险起见。第 200 行设置管理节点中的标志，以表示缓冲区已被释放。第 201 行将被释放缓冲区的管理节点放入内存池的空闲链表中。

22.2.2.5 模块管理

当需要查看所有内存池的状态时可以调用 mpool_dump()函数，其实现如图 22.44 所示。

```
00207: static bool mpool_dump_for_each (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00208: {
00209:     mpool_handle_t handle = (mpool_handle_t)_p_node;
00210:
00211:     UNUSED (_p_dll);
00212:     UNUSED (_p_arg);
00213:
00214:     console_print (" Name: %s\n", handle->name_);
00215:     console_print (" Buffer Size: %u\n", handle->buffer_size_);
00216:     console_print (" Capacity: %u\n", handle->buffer_count_);
00217:     console_print (" Available: %u\n", dll_size (&handle->free_buffer_));
00218:     console_print (" No Buffer: %u\n", handle->stats_nobuf_);
00219:     console_print ("\n");
00220:     return true;
00221: }
00222:
```

```
00223: void mpool_dump ()
00224: {
00225:     if (is_in_interrupt ()) {
00226:         return;
00227:     }
00228:
00229:     scheduler_lock ();
00230:     console_print ("Summary\n");
00231:     console_print ("-------\n");
00232:     console_print (" Supported: %u\n", CONFIG_MAX_MPOOL);
00233:     console_print (" Allocated: %u\n", dll_size (&g_used_mpool));
00234:     console_print (" .BSS Used: %d\n", ((usize_t)&g_used_mpool -
00235:         (usize_t)&g_mpool_pool [0]) + sizeof (g_used_mpool));
00236:     console_print ("\n");
00237:     console_print ("Pool Details\n");
00238:     console_print ("------------\n");
00239:     (void) dll_traverse (&g_used_mpool, mpool_dump_for_each, 0);
00240:     console_print ("\n");
00241:     scheduler_unlock ();
00242: }
```

图 22.44

module_mpool()函数是内存池管理模块的模块回调函数，它只关心在系统终止化时是否存在内存池没有被删除，以帮助发现潜在的内存泄漏问题。其实现如图 22.45 所示。

```
00244: static bool mpool_check_for_each (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00245: {
00246:     mpool_handle_t handle = (mpool_handle_t)_p_node;
00247:
00248:     UNUSED (_p_dll);
00249:     UNUSED (_p_arg);
00250:
00251:     console_print ("Error: memory pool \"%s\" isn't deleted\n", handle->name_);
00252:     return true;
00253: }
00254:
00255: error_t module_mpool (system_state_t _state)
00256: {
00257:     if (STATE_DESTROYING == _state) {
00258:         (void) dll_traverse (&g_used_mpool, mpool_check_for_each, 0);
00259:     }
00260:     return 0;
00261: }
```

图 22.45

mpool 算法的实现有两个特点。其一，缓冲区的内存与管理节点的内存是通过定义数组的形式进行分配的，而不是通过动态分配的方式；其二，采用将缓冲区内存与管理节点内存完全分开的方式，以减小当出现缓冲区溢出时破坏管理信息的可能性。

由于缓冲区的分配与释放函数是采用开关中断的方式防止竞争问题的，因此可以在中断状态下使用这两个函数。

22.2.3 缓冲区泄漏检测

缓冲区泄漏检测仍可以采用 mblock 中所介绍的使用位置标识的形式。当然，可以考虑将堆和内存池的位置标识分别定义在不同的头文件中，以减小两个模块的耦合性。

为 mpool 模块增加缓冲区泄漏检测的改进不打算在本书中进行介绍，而是留给读者作为练习。

22.3 小结

内存管理分为两个大类，本章分别称之为堆管理和内存池管理。从堆中分配内存是按用户所需大小进行的，而内存池是采用固定缓冲区大小的方式。

堆空间会因为频繁的内存分配与释放而产生内存碎片，而内存碎片的数量将影响内存分配和释放的效率，乃至可能造成无法从碎片中分配出所需内存的情形。与之相反的是，内存池方法不存在内存碎片，且内存的分配和释放速度是恒定的。

在现实的嵌入式系统中，通常会结合使用堆分配和内存池分配两种方法来实现产品的功能。对于分配速度要求高且所需分配的内存块大小相对恒定的情形下，内存池是首选方法。如果所需分配内存块的大小存在较大的差异，则采用从堆中分配内存的方式更可取，因为它更能节约内存资源。



练习与思考

1. 在 mblock 堆管理实现中，尽管采用了增加数据校验的方式来保证 mblock_t 和 mhead_t 数据结构的完整性，但采用这种方式能百分之百地保证完整性吗？为了回答这一问题，读者需要很仔细地阅读源代码，并找出其中存在的漏洞及进一步的改进方法。

2. 如何防止内存池中的缓冲区出现写溢出？读者有怎样的设计想法？



第 23 章 设备管理，方便与外设交互

设备管理（device management）模块通过对各种各样的硬件资源进行抽象，以一种更容易理解和使用的形式展现出来。抽象的结果是通过引入一定的设备管理模型，然后将硬件资源纳入模型之下，并让驱动程序（driver）作为模型与真实硬件间的桥梁。引入设备管理模块的好处是，一旦理解了它就可以举一反三地与各种硬件资源实现交互。

在计算机的世界里存在各种各样具有特定独立功能的硬件资源。实时时钟 RTC、UART 串口、以太网接口等，都是不同类型的硬件资源。在一个系统中也可以存在多个同类硬件，比如采用同样芯片的以太网接口在系统中可以有多个。由此看来，硬件资源存在“类”与“个”之分，且“个”是“类”的实例。类又存在大类与小类之别，比如以太网接口就是一个大类，但以太网接口又可以通过不同厂商生产的芯片来实现，而每一种型号的芯片就形成了小类。

根据设备对数据的组织特点，可以将设备分成三大类：字符设备、块设备和帧设备（有的称为网络设备）。字符设备的数据以字符流形式被访问，如串口、键盘。块类型设备能够随机访问固定大小的数据片，如硬盘、闪存。帧设备是一帧一帧发送和接收数据包的，如以太网接口。

三大类设备需要我们提供不同的模型来管理它们。其中字符设备的管理最简单，目前 ClearRTOS 只实现了字符设备管理，所以本章后面只探讨字符设备管理。块设备通常与文件系统紧密相关，帧设备则与（网络）协议栈相关，这两大块在现有的 ClearRTOS 中都不涵盖。

23.1 字符设备管理

对字符设备的抽象可以从与之交互的动作着手，动作包括打开（open）、关闭（close）、读（read）、写（write）和控制（control）。其中打开操作可以理解为对硬件资源进行使用前的初始化，关闭操作则进行相反的操作。如果每一个动作都对应一个函数，则为某一设备编写驱动程序就是实现这五个函数。

对硬件资源我们可通过将其共性与个性进行分离的方式来进行管理。共性表现为硬件的型号，同一型号设备的操控完全相同。表达共性是通过驱动程序来完成的，也就是说，对于每一

型号的硬件资源都存在一个与之对应的驱动程序。

硬件资源的个性表现在其拥有不同于其他个体的属性。比如说，一个系统中由两块完全相同的芯片所实现的串口，由于它们的芯片型号是一样的，所以使用同样的驱动程序，但是每块芯片所占用的中断线或 I/O 地址空间将完全不同，这两个不同的属性体现了两个串口的个性。

设备的个性很容易想到用数据结构（中的不同值）进行表达。当调用设备驱动程序（的五个函数）时，通过使用不同的用于表达设备个性的数据结构实例作为参数，以此实现共性与个性的相结合。这也正是本章设备管理模块的设计思想。

一个设备一定拥有其独立的属性，包括：

- I/O 地址空间。处理器对外部硬件资源的控制和访问必须通过对外部硬件内的寄存器的读和写来实现，为了实现对外部硬件资源的控制和访问，各外部硬件将在处理器的 I/O 空间中占据不同的位置。
- 硬件中断。不少硬件是通过中断的形式告知处理器取走硬件收到的数据或要求提供所需发送的数据。
- 内存。有些硬件需要提供预先分配好的、专用的内存，以便硬件能使用它来保存收到的数据或缓冲将要发送的数据。
- 同步资源。为了防止出现竞争问题，一个在多任务环境中的设备需要采用一定的同步手段在多任务间进行同步处理。
- 设备名称。这是为了我们能方便地对其进行标识，所以设备名称不能重复。

设备管理所需要完成的工作，就是实现对驱动程序和设备的有效组织，进而对外展现为一定的模型并提供统一的函数进行设备访问。对字符设备进行访问需要五个函数：

- device_open()函数。
- device_close()函数。
- device_read()函数。
- device_write()函数。
- device_control()函数。

由于每个设备都有名字，我们可以通过指定设备名称调用 device_open()函数打开一个设备。设备一旦被成功打开，device_open()函数将返回设备标识，我们可以通过这个标识来调用其他四个操作函数。

那驱动程序与每个设备又如何组织呢？这可以借助目录结构来实现，图 23.1 通过目录结构的形式表示了两大类设备的驱动程序和五个设备。其中三个设备属于时钟，而另外两个设备属于串口。

从图中也可以看出，“/dev/”是整个设备管理树的根，下一级是驱动程序，设备是挂在驱动程序之下的。如果要打开一个设备，在调用 device_open()函数时需要指明其路径全名。比如，对于“滴答”设备其全名为“/dev/clock/tick”。

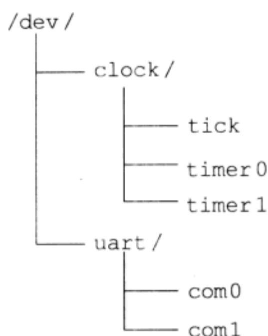


图 23.1

这棵树在系统初始化时，通过先安装驱动程序，然后注册设备的方式创建。比如，对于三个时钟设备，先要以“/dev/clock”为名安装驱动程序，然后分别以“/dev/clock/tick”、“/dev/clock/timer0”和“/dev/clock/timer1”三个名称注册设备。

23.2　中断管理

由于操作系统自身的实现需要频繁地打开和关闭中断，因此对于中断的管理不大适合将其抽象成一个设备，而是直接了当地提供相应的操作函数更合适。由于中断管理与设备管理有较强的关联度，所以将这个小节放入了本章中。

23.2.1　中断向量表

当一个中断发生时，需要调用对应的中断服务程序以处理中断事件。尽管有的处理器本身存在对中断服务程序的要求，比如，有的处理器规定好了各中断所在的地址空间，以及每个中断服务程序的大小，但我们仍可以采用一个数组来存放所有的中断服务程序的函数指针，这样的数组就是软件层面的中断向量表。有了软件层面的中断向量表之后，可以定义一个公共的函数，这个函数在中断发生时直接被调用，函数内部通过中断号从软件中断向量表中找到真正的中断服务程序。图 23.2 是与软件层面的中断向量表相关的程序定义。图中的 g_interrupt_vector 数组承担着表达中断向量表的功能，它被用于存放所有中断服务函数的指针。

```
00038: #define CONFIG_MAX_INTERRUPT        64
00039: #define INTERRUPT_LAST_INDEX        (CONFIG_MAX_INTERRUPT - 1)
00040: #define INTERRUPT_NONE               (-1)
00047: typedef void (*interrupt_handler_t) (int _vector);
00048:
```

```
00033: static interrupt_handler_t g_interrupt_vector [CONFIG_MAX_INTERRUPT];
```

图 23.2

在目前的ClearRTOS中，中断是通过使用Linux操作系统中的信号（signal）来模拟的：当在Linux操作系统上启动的定时器到期时，会产生一个SIGALRM信号；当我们在运行程序的终端上按下“Ctrl+C”组合键时，就会产生一个SIGINT信号；等等。通过使用Linux操作系统中的信号相关函数，可以使得每一个信号发生时调用同一个处理函数，即图23.3中的signal_handler()函数。该函数的_signal参数用于指示发生信号的标识，在此也可以将其理解为（模拟）中断号，其值由操作系统在调用函数时指定。

```
embedded/code/platform/arch/simulator/src/interrupt.c
00076: static void signal_handler (int _signal)
00077: {
00078:     g_interrupt_vector [_signal] (_signal);
00079: }
00080:
00081: static void signal_init ()
00082: {
00083:     struct sigaction action;
00084:     sigset_t mask;
00085:
00086:     mask = g_signal_mask;
00087:     (void) sigprocmask (SIG_UNBLOCK, &mask, 0);
00088:
00089:     action.sa_handler = signal_handler;
00090:     action.sa_mask = mask;
00091:     action.sa_flags = SA_RESTART;
00092:     (void) sigaction (SIGINT, &action, 0);
00093:     (void) sigaction (SIGQUIT, &action, 0);
00094:     (void) sigaction (SIGKILL, &action, 0);
00095:     (void) sigaction (SIGALRM, &action, 0);
00096:     (void) sigaction (SIGIO, &action, 0);
00097:     (void) sigaction (SIGSTOP, &action, 0);
00098: }
```

图23.3

图中的signal_init()函数将ClearRTOS所关心信号的处理函数设置为signal_handler()。当一个信号发生时，公共处理函数signal_hanlder()会被调用，它通过_signal参数从g_interrupt_vector数组中找到对应的处理函数进行处理。

23.2.2 中断控制

在Linux操作系统上运行的程序，它对信号是否做出反应是可以通过函数控制的。通过阻塞（或屏蔽）信号的方式，可以使程序永远收不到相应的信号。比方说，如果阻塞SIGINT信号，当程序运行时我们在终端上按下“Ctrl+C”组合键就无法终止程序的运行。这个特性正好可以模拟处理器中断的开与关。图23.4示例说明了中断控制函数的实现。

```
00071: static inline void global_interrupt_enable (interrupt_level_t _level)
00072: {
00073:     int status;
00074:
00075:     extern sigset_t g_signal_mask;
```

```
00076:
00077:     if (INTERRUPT_ENABLED == _level) {
00078:         status = sigprocmask (SIG_UNBLOCK, &g_signal_mask, 0);
00079:     }
00080:     else {
00081:         status = sigprocmask (SIG_BLOCK, &g_signal_mask, 0);
00082:     }
00083:     if (status) {
00084:         printf ("Error: global_interrupt_enable() is failed\n");
00085:     }
00086: }
00087:
00088: static inline interrupt_level_t global_interrupt_disable ()
00089: {
00090:     int status;
00091:     sigset_t old_mask;
00092:
00093:     extern sigset_t g_signal_mask_empty;
00094:     extern sigset_t g_signal_mask;
00095:
00096:     (void) sigemptyset (&old_mask);
00097:     status = sigprocmask (SIG_BLOCK, &g_signal_mask, &old_mask);
00098:     if (status) {
00099:         printf ("Error: global_interrupt_disable() is failed\n");
00100:     }
00101:     if (memcmp ((void *)&g_signal_mask_empty, (void *)&old_mask,
00102:         sizeof(sigset_t))) {
00103:         return INTERRUPT_DISABLED;
00104:     }
00105:     return INTERRUPT_ENABLED;
00106: }
00107:
00108: static inline interrupt_level_t interrupt_level_get ()
00109: {
00110:     sigset_t old_mask;
00111:
00112:     extern sigset_t g_signal_mask_empty;
00113:
00114:     (void) sigemptyset (&old_mask);
00115:     (void) sigprocmask (SIG_BLOCK, 0, &old_mask);
00116:
00117:     if (memcmp ((void *)&g_signal_mask_empty, (void *)&old_mask,
00118:         sizeof(sigset_t))) {
00119:         return INTERRUPT_DISABLED;
00120:     }
00121:     return INTERRUPT_ENABLED;
00122: }
```

embedded/code/platform/[illegible]/src/interrupt[illegible]

```
00124: void interrupt_enable (int _vector)
00125: {
00126:     UNUSED (_vector);
00127:     // do nothing on Linux for simplifying
00128: }
00129:
00130: void interrupt_disable (int _vector)
00131: {
00132:     UNUSED (_vector);
00133:     // do nothing on Linux for simplifying
00134: }
```

图 23.4

图中所示 interrupt.h 中的三个函数，用于模拟对处理器全局中断的开关控制和获取状态。一个信号对应于一位掩码，开中断就是让操作系统不要阻塞信号，反之阻塞信号。interrupt.c 中的两个函数可以用于控制单个信号的阻塞与非阻塞，或者说可以用它来模拟对单个中断的开与关。从模拟的角度来看，ClearRTOS 目前并不需要对单个中断进行开关控制，所以这两个函数的实现是空的。

23.2.3　中断状态管理

中断一旦发生，需要通过设计来跟踪处理器是否处于中断状态。由于中断是可以嵌套的（参见 1.6 节），因此需要使用整型变量跟踪中断的嵌套级数。图 23.5 示例说明了相关实现。

```
00032: static int g_interrupt_nested_count;
00035: static interrupt_exit_callback_t g_interrupt_exit_callback;
00036:
00037: void interrupt_enter ()
00038: {
00039:     interrupt_level_t level;
00040:
00041:     level = global_interrupt_disable ();
00042:     g_interrupt_nested_count ++;
00043:     global_interrupt_enable (level);
00044: }
00045:
00046: void interrupt_exit ()
00047: {
00048:     interrupt_level_t level;
00049:
00050:     level = global_interrupt_disable ();
00051:     g_interrupt_nested_count --;
00052:     global_interrupt_enable (level);
00053:     // give an opportunity to the scheduler for task switch
00054:     if (0 == g_interrupt_nested_count && 0 != g_interrupt_exit_callback) {
00055:         g_interrupt_exit_callback ();
00056:     }
00057: }
00058:
00059: bool is_in_interrupt ()
00060: {
00061:     return (bool) (g_interrupt_nested_count != 0);
00062: }
00063:
00105: void interrupt_exit_callback_install (interrupt_exit_callback_t _handler)
00106: {
00107:     g_interrupt_exit_callback = _handler;
00108: }
```

图 23.5

第 32 行定义的整型变量用于跟踪中断的嵌套级数。当每一个中断发生时，需要确保在中断服务程序中调用 interrupt_enter()和 interrupt_exit()函数，以便通过加减这个变量的方式记录中断的嵌套级数。一旦变量的值为 0，就说明处理器是处于非中断状态的。

在讲解任务的章节中指出，当处理器退出中断状态时需要触发一次任务切换操作。中断管理模块通过使用回调函数的方式减小与任务管理模块间的耦合，这正是第 35 行 g_interrupt_exit_callback 变量的作用，该变量记录的是回调函数的指针。任务管理模块以可通过调用 interrupt_exit_callback_install()函数实现回调函数的安装。当 interrupt_exit()函数被调用时，第 54 行检查中断的嵌套级数是否为 0，以及回调函数是否已设置，如果两个判断条件的结果是 true，则在第 55 行调用该回调函数，从而在中断状态返回的过程中实现任务切换。前面指出了，由于 ClearRTOS 不能接管 Linux 操作系统的中断，所以并不能实现这个功能。

23.2.4　设备与中断

在一个存在设备管理的操作系统中，中断的归属很明了——每个中断都应当属于某个（或几个）设备。

中断产生时，interrupt_handler()函数（实现如图 23.6 所示）将被调用，因为这个函数的指针在系统初始化时会被赋值给 g_interrupt_vector 数组中的每一个元素。interrupt_handler()函数的第一步是调用 interrupt_enter()函数表示处理器进入了中断状态；然后在第 68 行调用设备管理模块所提供的回调函数；最后通过调用 interrupt_exit()函数表示处理器要退出当前的中断状态。

```
00034: static interrupt_handler_t g_device_interrupt_handler;
00035:
00064: static void interrupt_handler (int _vector)
00065: {
00066:     interrupt_enter ();
00067:     if (g_device_interrupt_handler != null) {
00068:         g_device_interrupt_handler (_vector);
00069:     }
00070:     else {
00071:         // oops! log error? hold on, we are in interrupt!
00072:     }
00073:     interrupt_exit ();
00074: }
00099:
00100: void device_interrupt_handler_install (interrupt_handler_t _handler)
00101: {
00102:     g_device_interrupt_handler = _handler;
00103: }
```

图 23.6

g_device_interrupt_handler 变量用于记录设备管理模块的回调函数。当中断发生时，中断服务程序需要调用该回调函数通知设备管理模块，以便找到中断所对应的设备处理该中断。设备管理模块的回调函数可以通过 device_interrupt_handler_install()函数进行设置。

23.2.5　模块管理

中断管理模块的初始化和终止化实现位于 module_interrupt()函数中，其实现如图 23.7 所示。

```
embedded/code/platform/arch/simulator/src/interrupt.c
00136: error_t module_interrupt (system_state_t _state)
00137: {
00138:     if (STATE_INITIALIZING == _state) {
00139:         for (int vector = 0; vector <= INTERRUPT_LAST_INDEX; vector ++) {
00140:             g_interrupt_vector [vector] = interrupt_handler;
00141:         }
00142:         (void) sigemptyset (&g_signal_mask_empty);
00143:         (void) sigfillset (&g_signal_mask);
00144:         g_interrupt_nested_count = 0;
00145:         signal_init ();
00146:         global_interrupt_enable (INTERRUPT_ENABLED);
00147:     }
00148:     else if (STATE_DESTROYING == _state) {
00149:         global_interrupt_enable (INTERRUPT_DISABLED);
00150:     }
00151:     return 0;
00152: }
```

图 23.7

在系统的初始化阶段，第 139～146 行的代码将被执行，这段代码将 interrupt_handler()函数设置为所有中断的服务程序。第 142 和 143 行对用于控制信号的变量进行初始化。第 145 行调用 signal_init()函数对（模拟中断的）信号进行初始化。第 146 行让整个系统的中断处于开启状态。需要终止系统时，在第 149 行关闭整个系统的中断。

23.3　实现设备管理

设备管理实现中需要表达的第一个概念是设备，其数据结构如图 23.8 所示。

```
embedded/code/platform/device/inc/device.h
00034: typedef u32_t device_mode_t;
00035: typedef u32_t open_mode_t;
00036: typedef int control_option_t;
00037: typedef struct type_device device_t, *device_handle_t;
00038:
00057: struct type_device {
00058:     dll_node_t node_;
00059:     driver_handle_t driver_;
00060:     magic_number_t magic_number_;
00061:     char name_ [NAME_MAX_LENGTH + 1];
00062:     // below variables should be initialized by each driver
00063:     int interrupt_vector_;
00064:     void (*interrupt_handler_) (device_handle_t _handle);
00065: };
```

图 23.8

其中的第 57～65 行定义了设备的管理数据结构，在第 37 行将其重新定义为 device_t 类型和 device_handle_t。第 58 行的 node_变量用做链表节点。第 59 行的 driver_变量指向设备所对应的驱动程序。第 60 行的 magic_number_用于标识设备数据结构是否合法。第 61 行的 name_变量用于记录设备的名称。第 63 行的 interrupt_vector_变量用于记录设备的中断向量号，第 64

行的 interrupt_handler_指向的是设备的中断服务程序。

图 23.9 示例说明了设备管理中的另外两个重要的数据结构，它们都与驱动程序有关。

embedded/code/platform/device/inc/device.h

```
00039: typedef struct {
00040:     error_t (*open_) (device_handle_t _handler, open_mode_t _mode);
00041:     error_t (*close_) (device_handle_t _handler);
00042:     error_t (*read_) (device_handle_t _handler, void *_buf, usize_t _size);
00043:     error_t (*write_) (device_handle_t _handler, const void *_buf, usize_t _size);
00044:     error_t (*control_) (device_handle_t _handle, control_option_t _option,
00045:         int _int_arg, void *_ptr_arg);
00046: } device_operation_t;
00047:
00048: typedef struct {
00049:     dll_t devices_;
00050:     device_operation_t operation_;
00051:     device_mode_t mode_;
00052:     magic_number_t magic_number_;
00053:     int count_opened_;
00054:     char name_ [NAME_MAX_LENGTH*2 + 1];
00055: } driver_t, *driver_handle_t;
```

图 23.9

第 39～46 行所定义的 device_operation_t 数据结构用于记录设备的五个操作函数。第 48～55 行是驱动程序的数据结构。一个驱动程序可以对应多个设备，所有的设备是通过链表的形式挂接在第 49 行的 devices_链表中的。第 50 行的 operation_变量用于记录设备的五个操控函数。第 51 行的 mode_变量用于记录这类设备的模式，在现有的 ClearRTOS 中这个变量并没有被使用。第 53 行定义的 count_opened_变量用于记录在一个驱动上有多少个设备被打开了。在系统关闭时，可以通过检测这个变量判断是否存在设备没有关闭的情形，以帮助发现潜在的资源泄漏。第 54 行的 name_变量用于保存驱动程序的名称。

图 23.10 是其他的定义。第 41～44 行定义的 device_statistics_t 数据结构用于管理模块的统计信息，第 49 行定义了该结构的一个实例。第 46 行定义了 g_driver_pool 驱动池，所能安装的最大驱动数由 CONFIG_MAX_DRIVER 宏指定。第 47 行定义的 g_interrupt_map 数组用于实现中断号与设备的映射。

embedded/code/platform/device/src/device.c

```
00032: #define CONFIG_MAX_DRIVER          8
00033: #define DRIVER_LAST_INDEX          (CONFIG_MAX_DRIVER - 1)
00034: #define MAGIC_NUMBER_DRIVER        0x44524956L
00035: #define MAGIC_NUMBER_DEVICE        0x44455649L
00036:
00037: #define is_invalid_handle(_handle) (((_handle) == null) || \
00038:     ((_handle)->magic_number_ != MAGIC_NUMBER_DEVICE) || \
00039:     ((_handle)->driver_->magic_number_ != MAGIC_NUMBER_DRIVER))
00040:
00041: typedef struct {
00042:     statistic_t stats_invalid_index_;
00043:     statistic_t stats_invalid_binding_;
00044: } device_statistics_t;
```

```
00045:
00046: static driver_t g_driver_pool [CONFIG_MAX_DRIVER];
00047: static device_handle_t g_interrupt_map [CONFIG_MAX_INTERRUPT];
00048: //lint -esym(728, g_statistics) -esym(551, g_statistics)
00049: static device_statistics_t g_statistics;
```

图 23.10

23.3.1　安装驱动程序

驱动程序的安装是通过 driver_install()函数完成的，实现如图 23.11 所示。

```
00110: error_t driver_install (const char _name [],
00111:     const device_operation_t *_p_operation, device_mode_t _mode)
00112: {
00113:     int idx;
00114:     driver_handle_t p_driver = null;
00115:     const char *dev_root = "/dev/";
00116:
00117:     if ((null == _name) || (strncmp (_name, dev_root, strlen (dev_root)) != 0))
{
00118:         // a driver has no name?
00119:         return ERROR_T (ERROR_DRIVER_INVNAME);
00120:     }
00121:     if (null == _p_operation || null == _p_operation->open_ ||
00122:         null == _p_operation->close_) {
00123:         return ERROR_T (ERROR_DRIVER_INVOPT);
00124:     }
00125:     // a driver only can be installed before system is UP
00126:     if (system_state () > STATE_INITIALIZING) {
00127:         return ERROR_T (ERROR_DRIVER_INVSTATE);
00128:     }
00129:     // check whether the driver is already installed
00130:     for (idx = 0; idx <= DRIVER_LAST_INDEX; idx ++) {
00131:         if ((MAGIC_NUMBER_DRIVER == __driver_pool [idx].magic_number_) &&
00132:             (0 == strncmp (g_driver_pool [idx].name_,_name, strlen (_name)))) {
00133:             return ERROR_T (ERROR_DRIVER_INSTALLED);
00134:         }
00135:     }
00136:
00137:     for (idx = 0; idx <= DRIVER_LAST_INDEX; idx ++) {
00138:         if (g_driver_pool [idx].magic_number_ != MAGIC_NUMBER_DRIVER) {
00139:             p_driver = &g_driver_pool [idx];
00140:             break;
00141:         }
00142:     }
00143:     if (null == p_driver) {
00144:         return ERROR_T (ERROR_DRIVER_NODRV);
00145:     }
00146:     p_driver->magic_number_ = MAGIC_NUMBER_DRIVER;
00147:     p_driver->operation_ = *_p_operation;
00148:     p_driver->mode_ = _mode;
00149:     strncpy (p_driver->name_, _name, sizeof (p_driver->name_) - 1);
00150:     p_driver->name_ [sizeof (p_driver->name_) - 1] = 0;
00151:     dll_init (&p_driver->devices_);
00152:     return 0;
00153: }
```

图 23.11

从函数的实现来看，它需要三个参数。第一个参数用于指明驱动名称；第二个参数指定设备的五个操控函数；最后一个参数用于设置驱动模式，目前没有被使用。

第 117～120 行先检查驱动名称是否符合要求，如果不符合要求就返回错误。第 121～124 行检查所提供的操控函数是否有效。很明显一个设备的驱动必须包含打开和关闭两个操作，因此这两个操作函数指针必须有效。第 126～128 行用于限制驱动程序只能在整个系统进入“已启动（STATE_UP）”状态之前被安装。第 130～135 行确保被安装的驱动是第一次安装。第 137～142 行从驱动池中找到一个空闲的存放位置。第 143～145 行如果发现无空闲位置可用则返回错误。第 147～150 行根据输入参数初始化驱动的其他信息。第 151 行的目的是初始化驱动程序的设备列表。

23.3.2　注册设备

通过调用 device_register()函数完成对设备的注册，该函数实现如图 23.12 所示。

```
00155: error_t device_register (const char _name [], device_handle_t _handle)
00156: {
00157:     int idx;
00158:     driver_handle_t p_driver = null;
00159:     const char *p_name;
00160:
00161:     if (null == _name) {
00162:         // a device has no name?
00163:         return ERROR_T (ERROR_DEVICE_REGISTER_INVNAME);
00164:     }
00165:     if (null == _handle) {
00166:         return ERROR_T (ERROR_DEVICE_REGISTER_INVHANDLE);
00167:     }
00168:     // a device only can be registered before system is UP
00169:     if (system_state () > STATE_INITIALIZING) {
00170:         return ERROR_T (ERROR_DRIVER_INVSTATE);
00171:     }
00172:
00173:     // find the driver for the device
00174:     for (idx = 0; idx <= DRIVER_LAST_INDEX; idx ++) {
00175:         if ((MAGIC_NUMBER_DRIVER == g_driver_pool [idx].magic_number_) &&
00176:         (0 == strncmp (g_driver_pool [idx].name_, _name,
00177:             strlen (g_driver_pool [idx].name_)))) {
00178:             p_driver = &g_driver_pool [idx];
00179:             break;
00180:         }
00181:     }
00182:     if (null == p_driver) {
00183:         return ERROR_T (ERROR_DEVICE_REGISTER_NODRV);
00184:     }
00185:     if (_handle->interrupt_vector_ != INTERRUPT_NONE) {
00186:         g_interrupt_map [_handle->interrupt_vector_] = _handle;
00187:     }
00188:     _handle->driver_ = p_driver;
00189:     // strip the driver name from device name before passing it into
00190:     // the device open () operation
00191:     p_name = &_name [strlen (p_driver->name_)];
```

```
00192:     strncpy (_handle->name_, p_name, sizeof (_handle->name_) - 1);
00193:     _handle->name_ [sizeof (_handle->name_) - 1] = 0;
00194:     _handle->magic_number_ = MAGIC_NUMBER_DEVICE;
00195:     dll_push_tail (&p_driver->devices_, &_handle->node_);
00196:     return 0;
00197: }
```

图 23.12

第 174～184 行根据设备名称找到对应的驱动程序。第 185～186 行检查如果被注册设备存在中断，则在中断与设备句柄的映射表中设置中断所对应的设备句柄。第 188 行为设备设置驱动程序。第 191～193 行初始化设备名称。注意，设备名称并不包含驱动名称部分。第 194 行对设备数据结构进行标识，以表明它是一个有效的设备数据结构。第 195 行将设备挂接到驱动程序的链表中。

g_interrupt_map 数组用于实现中断号与设备的映射。当设备初始化时，会将 device_interrupt_handler()函数注册到中断管理模块，一旦中断发生，该函数将被调用（参见 23.2.4 节），其实现如图 23.13 所示。

```
embedded/code/platform/device/src/device.c
00051: static void device_interrupt_handler (int _vector)
00052: {
00053:     if (_vector > INTERRUPT_LAST_INDEX) {
00054:         interrupt_level_t level = global_interrupt_disable ();
00055:         g_statistics.stats_invalid_index_ ++;
00056:         global_interrupt_enable (level);
00057:         return;
00058:     }
00059:     if (null == g_interrupt_map [_vector]) {
00060:         interrupt_level_t level = global_interrupt_disable ();
00061:         g_statistics.stats_invalid_binding_ ++;
00062:         global_interrupt_enable (level);
00063:         return;
00064:     }
00065: 
00066:     g_interrupt_map [_vector]->interrupt_handler_ (g_interrupt_map [_vector]);
00067: }
```

图 23.13

第 53～58 行先检查输入的中断号是否有效。第 59 行检查映射表中是否存在有效的设备句柄，当一个设备存在中断时它的句柄才会出现在映射表中。第 66 行调用设备句柄中所保存的中断服务程序进行真正的中断处理。

23.3.3　打开设备

要使用一个设备必须通过调用 device_open()函数打开，打开设备后才能对设备进行读、写和控制。设备打开函数的实现如图 23.14 所示。

embedded/code/platform/device/src/device.c

```
00199: static bool device_find (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00200: {
00201:     device_handle_t handle = (device_handle_t)_p_node;
00202:
00203:     UNUSED (_p_dll);
00204:     UNUSED (_p_arg);
00205:
00206:     // if find the device, return false to termniate traversal
00207:     if (0 == strcmp (_p_arg, handle->name_)) {
00208:         return false;
00209:     }
00210:     return true;
00211: }
00212:
00213: error_t device_open (device_handle_t *_p_handle,
00214:     const char _name [], open_mode_t _mode)
00215: {
00216:     int idx;
00217:     driver_handle_t p_driver = null;
00218:     const char *p_name;
00219:     device_handle_t handle;
00220:     error_t ecode;
00221:
00222:     if (null == _name) {
00223:         return ERROR_T (ERROR_DEVICE_OPEN_INVNAME);
00224:     }
00225:
00226:     // find the driver for the device
00227:     for (idx = 0; idx <= DRIVER_LAST_INDEX; idx ++) {
00228:         if ((MAGIC_NUMBER_DRIVER == g_driver_pool [idx].magic_number_) &&
00229:             (0 == strncmp (g_driver_pool [idx].name_, _name,
00230:             strlen (g_driver_pool [idx].name_)))) {
00231:             p_driver = &g_driver_pool [idx];
00232:             break;
00233:         }
00234:     }
00235:     if (null == p_driver) {
00236:         return ERROR_T (ERROR_DEVICE_OPEN_NODRV);
00237:     }
00238:     // strip the driver name from device name before passing it
00239:     // into the device open () operation
00240:     p_name = &_name [strlen (p_driver->name_)];
00241:     // find the device
00242:     handle = (device_handle_t) dll_traverse (
00243:         &p_driver->devices_,device_find, (void *)p_name);
00244:     if (0 == handle) {
00245:         return ERROR_T (ERROR_DEVICE_OPEN_NODEV);
00246:     }
00247:     // we don't need to consider race condition for calling driver's
00248:     // open () operation, intead, the specific device driver should be responsible.
00249:     ecode = p_driver->operation_.open_ (handle, _mode);
00250:     if (0 == ecode) {
00251:         interrupt_level_t level = global_interrupt_disable ();
00252:         p_driver->count_opened_ ++;
00253:         global_interrupt_enable (level);
00254:         *_p_handle = handle;
00255:     }
00256:     return ecode;
00257: }
```

图 23.14

当一个设备被成功打开时通过第一个参数返回；第二个参数用于指定要打开设备的名称；最后一个参数指示设备打开模式，在现有的 ClearRTOS 中没有被用到。

第 227～237 行通过设备名称找到设备对应的驱动程序。从 device_register()函数的实现来看，注册设备是挂接在驱动数据结构的链表中的，因此找到设备对应的驱动是打开设备的第一步。由于挂接在驱动之下的设备名称是不包括驱动名称的，因此在第 240 行让 p_name 变量指向不包含驱动名称的设备名部分，以便后面进行设备查找。第 242～246 行通过遍历驱动数据结构中的设备链表找到所需打开的设备，遍历链表所使用回调函数的实现位于第 199～211 行。

由于设备数据结构中存在设备特定的属性，而打开函数也正需要通过这些信息对硬件资源进行必要的初始化操作，所以设备对应的数据结构一旦找到，就在第 249 行调用驱动程序的打开函数，并将设备的数据结构作为第一个参数传递给它。设备一旦被成功打开，则在第 252 行更新打开计数，记录下有多少个设备在这个驱动下被打开。第 254 行将设备的句柄返回给函数调用者。

23.3.4 关闭设备

设备关闭函数 device_close()的实现如图 23.15 所示。

```
00259: error_t device_close (device_handle_t _handle)
00260: {
00261:     error_t ecode;
00262:
00263:     if (is_invalid_handle (_handle)) {
00264:         return ERROR_T (ERROR_DEVICE_CLOSE_INVHANDLE);
00265:     }
00266:
00267:     ecode = _handle->driver_->operation_.close_ (_handle);
00268:     if (0 == ecode) {
00269:         interrupt_level_t level = global_interrupt_disable ();
00270:         _handle->driver_->count_opened_ --;
00271:         global_interrupt_enable (level);
00272:     }
00273:     return ecode;
00274: }
```

图 23.15

第 263 行检查设备句柄是否有效。第 267 行调用驱动程序中的关闭函数对硬件资源进行关闭操作。设备关闭成功后在第 270 行更新统计计数。

23.3.5 设备读写与控制

设备读、写和控制操作分别对应 device_read()、device_write()和 device_control()函数，实现如图 23.16 所示。

```
00276: error_t device_read (device_handle_t _handle, void *_buf, usize_t _size)
00277: {
00278:     if (is_invalid_handle (_handle)) {
00279:         return ERROR_T (ERROR_DEVICE_READ_INVHANDLE);
00280:     }
00281:
00282:     if (null == _handle->driver_->operation_.read_) {
00283:         return ERROR_T (ERROR_DEVICE_READ_NOTSUPPORT);
00284:     }
00285:     return _handle->driver_->operation_.read_ (_handle, _buf, _size);
00286: }
00287:
00288: error_t device_write (device_handle_t _handle, const void *_buf, usize_t _size)
00289: {
00290:     if (is_invalid_handle (_handle)) {
00291:         return ERROR_T (ERROR_DEVICE_WRITE_INVHANDLE);
00292:     }
00293:
00294:     if (null == _handle->driver_->operation_.write_) {
00295:         return ERROR_T (ERROR_DEVICE_WRITE_NOTSUPPORT);
00296:     }
00297:     return _handle->driver_->operation_.write_ (_handle, _buf, _size);
00298: }
00299:
00300: error_t device_control (device_handle_t _handle,
00301:     control_option_t _option, int _int_arg, void *_ptr_arg)
00302: {
00303:     if (is_invalid_handle (_handle)) {
00304:         return ERROR_T (ERROR_DEVICE_CONTROL_INVHANDLE);
00305:     }
00306:
00307:     if (null == _handle->driver_->operation_.control_) {
00308:         return ERROR_T (ERROR_DEVICE_CONTROL_NOTSUPPORT);
00309:     }
00310:     return _handle->driver_->operation_.control_ (_handle,
00311:         _option, _int_arg, _ptr_arg);
00312: }
```

图 23.16

由于对硬件资源的读写和控制完全是由驱动程序完成的，因此这几个函数的实现都很简单，只是将调用转交给驱动程序的相应函数。

device_control()函数的参数需要额外提一下。_option 是指要对设备做怎样的控制，它可以由各设备独立定义。_int_arg 和_ptr_arg 分别是控制设备所需提供的两个（可选）参数，它的具体含义同样由各设备自行定义。

23.4　设备驱动程序实现

在 ClearRTOS 中存在多个设备，比如滴答、控制台(console)，以及用于终止程序运行的“Ctrl+C”功能也是以设备的形式存在于系统中的。下面通过介绍这些设备的驱动程序来让读者进一步掌握 ClearRTOS 中驱动程序的编写。

23.4.1 “滴答”设备

“滴答”在操作系统中的作用在 20.8 节中进行了说明。在真实的嵌入式系统中，“滴答”是通过使用处理器上的硬件定时器来实现的，而在目前的 ClearRTOS 中，“滴答”是通过使用 Linux 中的定时器实现的。

图 23.17 的第 36～40 行为“滴答”对应的数据结构定义，第 33～34 行是两个设备控制选项的定义。

```
embedded/code/platform/arch/x86/simulator/inc/clock.h
00033: #define OPTION_TICK_START        (((int)(MODULE_CLOCK) << 8) + 1)
00034: #define OPTION_TICK_STOP         (((int)(MODULE_CLOCK) << 8) + 2)
00035:
00036: typedef struct {
00037:     device_t common_;
00038:     bool is_opened_;
00039:     void (*tick_process_) ();
00040: } device_clock_t, *clock_handle_t;
```

图 23.17

每个设备的数据结构定义都必须包含一个类型为 device_t 的变量，且该变量必须放在数据结构的最开始处。第 38 行定义的布尔变量标识设备是否已打开。当“滴答”中断产生时需要调用由任务管理模块所提供的处理函数，该函数将被存放于第 39 行所定义的 tick_process_函数指针变量中。

每个设备都需要实现打开和关闭操作。由于“滴答”设备在这两个操作中没有实质性的工作，所以实现很简单。另外，“滴答”设备还需要控制函数，用来设置“滴答”的时间周期和控制启停。这三个函数的实现如图 23.18 所示。

```
embedded/code/platform/arch/x86/simulator/src/clock.c
00039: static error_t clock_open (device_handle_t _handle, open_mode_t _mode)
00040: {
00041:     clock_handle_t handle = (clock_handle_t)_handle;
00042:
00043:     UNUSED (_mode);
00044:
00045:     if (handle->is_opened_) {
00046:         return ERROR_T (ERROR_CLOCK_OPEN_OPENED);
00047:     }
00048:
00049:     handle->is_opened_ = true;
00050:     return 0;
00051: }
00052:
00053: static error_t clock_close (device_handle_t _handle)
00054: {
00055:     clock_handle_t handle = (clock_handle_t)_handle;
00056:
00057:     handle->is_opened_ = false;
00058:     return 0;
00059: }
00060:
```

```
00061: //lint -e{818}
00062: static error_t clock_control (device_handle_t _handle,
00063:     control_option_t _option, int _int_arg, void *_ptr_arg)
00064: {
00065:     clock_handle_t handle = (clock_handle_t)_handle;
00066:
00067:     switch (_option)
00068:     {
00069:     case OPTION_TICK_START:
00070:         {
00071:             struct itimerval tv;
00072:
00073:             //lint -e{611}
00074:             handle->tick_process_ = (interrupt_handler_t) _ptr_arg;
00075:             tv.it_value.tv_sec = 0;
00076:             tv.it_value.tv_usec = (long) _int_arg*1000;
00077:             tv.it_interval.tv_sec = 0;
00078:             tv.it_interval.tv_usec = (long) _int_arg*1000
00079:             (void) setitimer (ITIMER_REAL, &tv, 0);
00080:
00081:             interrupt_enable (_handle->interrupt_vector_);
00082:         }
00083:         break;
00084:     case OPTION_TICK_STOP:
00085:         {
00086:             struct itimerval tv;
00087:             struct sigaction act;
00088:
00089:             interrupt_disable (_handle->interrupt_vector_);
00090:
00091:             memset (&act, 0, sizeof(act));
00092:             act.sa_handler = SIG_IGN;
00093:             (void) sigaction (SIGALRM, &act, 0);
00094:
00095:             memset (&tv, 0, sizeof(tv));
00096:             (void) setitimer (ITIMER_REAL, &tv, 0);
00097:         }
00098:         break;
00099:     default:
00100:         return ERROR_T (ERROR_CLOCK_CONTROL_INVOPT);
00101:     }
00102:     return 0;
00103: }
```

图 23.18

clock_control()函数实现了两个控制选项。选项之一是启动定时器，其实现位于第 71～81 行。“滴答”是通过使用 Linux 操作系统的定时器实现的，当这一定时器到期时将产生 SIGALRM 信号，从 ClearRTOS 来看是一个模拟的中断。_int_arg 参数用于指定“滴答”的周期；_ptr_arg 参数用于指定处理“滴答”的回调函数。选项之二是停止定时器，其实现在第 86～96 行。

在实现了“滴答”的打开、关闭和控制三个函数后，需要实现安装驱动与注册设备的函数，这两个函数的实现如图 23.19 所示。

```
00033: void tick_interrupt_handler (device_handle_t _handle)
00034: {
```

```
00035:     clock_handle_t handle = (clock_handle_t)_handle;
00036:     handle->tick_process_ ();
00037: }
00038:
00105: error_t clock_driver_install (const char _name[])
00106: {
00107:     device_operation_t opt = {
00108:         .open_ = clock_open,
00109:         .close_ = clock_close,
00110:         .control_ = clock_control,
00111:     };
00112:     return driver_install (_name, &opt, 0);
00113: }
00114:
00115: error_t clock_device_register (const char _name [], clock_handle_t _handle)
00116: {
00117:     _handle->common_.interrupt_vector_ = SIGALRM;
00118:     _handle->common_.interrupt_handler_ = tick_interrupt_handler;
00119:     _handle->is_opened_ = false;
00120:     return device_register (_name, &_handle->common_);
00121: }
```

图 23.19

clock_driver_install()函数的实现不再多说。clock_device_register()函数的实现有两处需要留意。第一，设备标识是第二个输入参数，也就是说，“滴答”所对应的管理数据需要在外面定义；第二个需要注意的地方位于第 118 行，即需要设置相应的中断服务程序。tick_interrupt_handler()函数的实现同样可以从图 23.19 中找到，它直接调用设备所保存的回调函数。

在哪里调用 clock_driver_install()和 clock_device_register()函数将留到后面讲解。

23.4.2 控制台设备

控制台设备的驱动程序实现如图 23.20 和图 23.21 所示。其实现与前面的“滴答”设备在结构上一模一样。

```
00032: typedef struct {
00033:     device_t common_;
00034:     bool is_opened_;
00035: } device_console_t, *console_handle_t;
```

图 23.20

```
00035: static error_t console_open (device_handle_t _handle, open_mode_t _mode)
00036: {
00037:     console_handle_t handle = (console_handle_t)_handle;
00038:
00039:     UNUSED (_mode);
00040:
00041:     if (handle->is_opened_) {
00042:         return ERROR_T (ERROR_CONSOLE_OPEN_OPENED);
00043:     }
00044:
```

```
00045:     handle->is_opened_ = true;
00046:     return 0;
00047: }
00048:
00049: static error_t console_close (device_handle_t _handle)
00050: {
00051:     console_handle_t handler = (console_handle_t)_handle;
00052:
00053:     handle->is_opened_ = false;
00054:     return 0;
00055: }
00056:
00057: static error_t console_write (device_handle_t _handle,
00058:     const void *_buf,usize_t _size)
00059: {
00060:     UNUSED (_handle);
00061:
00062:     // write to the STDOUT for simulating a Console
00063:     write (0, _buf, _size);
00064:     return 0;
00065: }
00066:
00067: error_t console_driver_install (const char _name[])
00068: {
00069:     device_operation_t opt = {
00070:         .open_ = console_open,
00071:         .close_ = console_close,
00072:         .read_ = 0,
00073:         .write_ = console_write,
00074:         .control_ = 0
00075:     };
00076:
00077:     return driver_install (_name, &opt, 0);
00078: }
00079:
00080: error_t console_device_register (const char _name [], console_handle_t _handle)
00081: {
00082:     _handle->common_.interrupt_vector_ = INTERRUPT_NONE;
00083:     _handle->is_opened_ = false;
00084:     return device_register (_name, &_handle->common_);
00085: }
```

图 23.21

在现有的 ClearRTOS 中，控制台是通过 Linux 中的标准输出口进行模拟的。在真实的嵌入式系统中，控制台通常是一个串口。

对于模拟的控制台，我们只实现了打开、关闭和写操作的函数，而没有实现读和控制函数，在真实的嵌入式系统中，这两个函数同样必不可少。在 console.c 文件中，还实现了另外两个函数，其中 console_handle_set()函数用于设置控制台设备的句柄，console_print()函数用于向控制台输出可打印信息。两个函数的实现如图 23.22 所示。

```
00087: static device_handle_t g_console_handle;
00088:
00089: void console_handle_set (device_handle_t _handle)
00090: {
```

```
00091:     g_console_handle = _handle;
00092: }
00093:
00094: void console_print (const char* _format, ...)
00095: {
00096:     va_list arglist = 0;
00097:     static char buffer [1024];
00098:     static int length;
00099:     static int log_lost = 0;
00100:
00101:     va_start (arglist, _format);
00102:     length = vsnprintf (buffer, sizeof (buffer) - 1, _format, arglist);
00103:     va_end (arglist);
00104:     if (0 == g_console_handle) {
00105:         log_lost ++;
00106:     }
00107:     else {
00108:         (void) device_write (g_console_handle, buffer, (usize_t)length);
00109:         // log out a meesage to notify that there is log lost
00110:         if (log_lost != 0) {
00111:             length = snprintf (buffer, sizeof (buffer) - 1,
00112:                 "Warning: there is/are %d lines of log lost\n", log_lost);
00113:             log_lost = 0;
00114:         }
00115:     }
00116: }
```

图 23.22

第 87 行定义的 g_console_handle 变量用于记录控制台的句柄，以便在 console_print()函数中使用它，通过 console_handle_set()函数可以设置该变量的值。

console_print()函数的实现，也是通过使用 C 库函数的方式，将所需输出的消息格式化后放入到第 97 行定义的缓冲区中，然后调用 device_write()函数将缓冲区中的内容写入控制台设备中。

console_print()函数的实现需要特别注意一点。整个系统存在这样一种情况：在初始化时通过控制台 console_print()函数输出初始化信息，但此时有可能控制台设备还没有打开，这造成输出信息将无法显示在终端上。在这种情形下，通过使用第 99 行定义的 log_lost 变量，帮助记录有多少条信息在控制台设备被打开之前丢失了。一旦设备打开后，通过向控制台写一条告警信息，告知有多少条信息已丢失。控制台应当在绝大部分的模块初始化之前被打开。当在控制台出现有信息丢失的告警时，我们应当检查模块的初始化顺序是否合理，或者思考向控制台输出信息的时机是否恰当。

23.4.3　终止程序运行设备

在 Linux 操作系统中，“Ctrl+C”组合键被按下后，正在运行的程序将收到一个 SIGINT 信号，通过这个信号可以终止程序的运行。在 ClearRTOS 中我们可以使用这个组合键来优雅地终止程序。

为了保证程序的结构，在 ClearRTOS 中将“Ctrl+C”组合键终止程序的功能也当做是由一

个设备提供的，作者称为"终止程序运行设备"。有了前面两个设备的驱动编写经验后，终止程序设备的驱动程序也没有什么特别的地方，其实现如图 23.23 和图 23.24 所示。

embedded/code/platform/driver/ctrlc/inc/ctrlc.h

```
00033: typedef struct {
00034:     device_t common_;
00035:     bool is_opened_;
00036: } device_ctrlc_t, *console_ctrlc_t;
```

图 23.23

embedded/code/platform/driver/ctrlc/src/ctrlc.c

```
00034: void ctrlc_interrupt_handler (device_handle_t _handle)
00035: {
00036:     ctrlc_handle_t handle = (ctrlc_handle_t) _handle;
00037:
00038:     if (handle->is_opened_) {
00039:         console_print ("\nInfo: Ctrl+C pressed\n");
00040:         multitasking_stop ();
00041:     }
00042: }
00043:
00044: static error_t ctrlc_open (device_handle_t _handle, open_mode_t _mode)
00045: {
00046:     ctrlc_handle_t handle = (ctrlc_handle_t)_handle;
00047:
00048:     UNUSED (_mode);
00049:
00050:     if (handle->is_opened_) {
00051:         return ERROR_T (ERROR_CTRLC_OPEN_OPENED);
00052:     }
00053:
00054:     handle->is_opened_ = true;
00055:     interrupt_enable (_handle->interrupt_vector_);
00056:     console_print ("\nInfo: press Ctrl+C to terminate!\n");
00057:     return 0;
00058: }
00059:
00060: static error_t ctrlc_close (device_handle_t _handle)
00061: {
00062:     ctrlc_handle_t handle = (ctrlc_handle_t)_handle;
00063:
00064:     interrupt_disable (_handle->interrupt_vector_);
00065:     handle->is_opened_ = false;
00066:     return 0;
00067: }
00068:
00069: error_t ctrlc_driver_install (const char _name[])
00070: {
00071:     device_operation_t opt = {
00072:         .open_ = ctrlc_open,
00073:         .close_ = ctrlc_close,
00074:         .read_ = 0,
00075:         .write_ = 0,
00076:         .control_ = 0
00077:     };
00078:
00079:     return driver_install (_name, &opt, 0);
```

```
00080: }
00081:
00082: error_t ctrlc_device_register (const char _name [], ctrlc_handle_t _handle)
00083: {
00084:     _handle->common_.interrupt_vector_ = SIGINT;
00085:     _handle->common_.interrupt_handler_ = ctrlc_interrupt_handler;
00086:     _handle->is_opened_ = false;
00087:     return device_register (_name, &_handle->common_);
00088: }
```

图 23.24

当设备被打开时会在运行程序的终端上显示“Info: press Ctrl+C to terminate!”。当“Ctrl+C”组合键被按下后，ctrlc_interrupt_handler()函数会被调用，它通过调用 multitasking_stop()函数终止程序的运行。

23.5 驱动安装与设备注册

有了设备的驱动程序后，需要在合适的地方安装驱动程序及注册设备。和其他模块一样，设备管理模块也需要实现用于模块管理的回调函数，它就是 module_device()函数，其实现如图 23.25 所示。驱动程序的安装与设备注册可以以它作为一个切入点。

```
00069: error_t module_device (system_state_t _state)
00070: {
00071:     static device_handle_t console_handle;
00072:
00073:     if (STATE_INITIALIZING == _state) {
00074:         static device_console_t g_device_console;
00075:         error_t ecode;
00076:
00077:         device_interrupt_handler_install (device_interrupt_handler);
00078:
00079:         // make the Console ready and open it as soon as possible
00080:         ecode = console_driver_install ("/dev/uart/");
00081:         if (ecode != 0) {
00082:             return ecode;
00083:         }
00084:         ecode = console_device_register ("/dev/uart/com0", &g_device_console);
00085:         if (ecode != 0) {
00086:             return ecode;
00087:         }
00088:         ecode = device_open (&console_handle, "/dev/uart/com0", 0);
00089:         if (ecode != 0) {
00090:             return ecode;
00091:         }
00092:         console_handle_set (console_handle);
00093:
00094:         (void) device_registration_main ();
00095:     }
00096:     else if (STATE_DESTROYING == _state) {
00097:         for (int idx = 1; idx <= DRIVER_LAST_INDEX; idx ++) {
00098:             if ((MAGIC_NUMBER_DRIVER != g_driver_pool [idx].magic_number_) ||
00099:                 (0 == g_driver_pool [idx].count_opened_)) {
```

```
00100:                continue;
00101:            }
00102:            console_print ("Error: device(s) on driver \"%s\" is/are
00103:                not closed\n",g_driver_pool [idx].name_);
00104:        }
00105:        (void) device_close (console_handle);
00106:    }
00107:    return 0;
00108: }
```

图 23.25

在系统的初始化阶段，第 74～94 行的代码会被运行。第 77 行通过调用 device_interrupt_handler_install()函数将 device_interrupt_handler()函数安装为中断管理模块收到中断时的处理函数。

第 80～91 行代码用于处理控制台设备。在第 74 行定义了一个控制台设备的数据结构实例，即 g_device_console 变量，它是一个静态变量。第 80 行安装控制台的驱动程序。第 84 行注册一个控制台设备。在注册控制台设备时，所使用的设备句柄正是第 74 行定义变量的指针。第 88 行打开控制台设备。第 92 行通过调用 console_handle_set()函数设置控制台句柄。在第 88 行打开控制台时，返回的设备句柄实际上正是第 74 行定义变量的地址值，之所以这么“绕一圈”完全是为了程序的结构。需要注意，由于整个系统在初始化时会使用到控制台，所以应尽可能早地打开它。

第 94 行调用 device_registration_main()函数（实现如图 23.26 所示）完成整个系统除了控制台设备之外其他设备的驱动安装和设备注册。

在系统终止化时，第 97～104 行通过查看各驱动程序上所打开的设备数是否为 0，以此检查是否有设备没有关闭。如果存在这种情形，第 102 行在终端上会打印出一条错误信息。注意，遍历驱动程序的管理数组时，是从索引为 1 的位置开始的，因为位置 0 中放置的是控制台的驱动程序，而此时控制台设备并未关闭，我们需要跳过对它的检查[①]。第 105 行以关闭控制台设备结束设备管理模块的终止活动。

图 23.26 是 device_registration_main()函数的实现。在第 36～37 行定义了用于代表设备的数据结构实例，第 47～64 行实现了各设备驱动程序的安装和设备注册。

```
00036: static device_clock_t g_device_tick;
00037: static device_ctrlc_t g_device_ctrlc;
00041:
00042: error_t device_registration_main ()
00043: {
00044:     error_t ecode;
00045:
00046:     // 1) for Tick
```

① 因为控制台驱动程序在整个系统中是第一个被安装的，所以它将占据索引为 0 的位置。

```
00047:     ecode = clock_driver_install ("/dev/clock/");
00048:     if (ecode != 0) {
00049:         return ecode;
00050:     }
00051:     ecode = clock_device_register ("/dev/clock/tick", &g_device_tick);
00052:     if (ecode != 0) {
00053:         return ecode;
00054:     }
00055:
00056:     // 2) for Ctrl+C
00057:     ecode = ctrlc_driver_install ("/dev/ui/");
00058:     if (ecode != 0) {
00059:         return ecode;
00060:     }
00061:     ecode = ctrlc_device_register ("/dev/ui/ctrlc", &g_device_ctrlc);
00062:     if (ecode != 0) {
00063:         return ecode;
00064:     }
00078:     return 0;
00079: }
```

图 23.26

再一次提醒，除了控制台设备外整个系统的设备注册都应当放入 device_registration_main() 函数中以保证软件的结构。

23.6 小结

设备管理是通过抽象提供管理模型的，管理模型的确定也决定了驱动程序如何编写。对于字符设备、块设备和帧设备需要提供不同的管理模型。

对于字符设备可以从动作的角度进行抽象，包括打开、关闭、读、写和控制。通过为这五个动作定义操作函数，能很好地统一操作设备的接口函数。这种统一除了让代码保持良好的可读性和可维护性外，也有助于工程师举一反三地掌握如何与各类字符设备进行交互。



练习与思考

1. 在图 23.22 所示的 console_print()函数的实现中，为什么需要定义 g_console_handle 变量，且提供设置该变量的函数 console_handle_set()？另外，console_print()函数为什么调用 device_write()函数而不是调用 console_write()函数？这样做的好处是什么？

2. 在 23.4.2 节中所介绍的控制台设备的驱动中，并没有考虑互斥处理问题，在 Linux 上运行有问题吗？为什么？对于一个真实嵌入式系统的控制台驱动程序，其实现方面应如何处理互斥问题？

第 24 章

定时器，程序闹钟

程序中的定时器就像我们生活中的闹钟，用来提醒软件世界中某件事情的发生。硬件定时器的数目很有限，大多处理器只提供 3 到 4 个硬件定时器，这对于复杂的应用是远远不够用的。在 23.4.1 节介绍的“滴答”设备，其实就对应于处理器的一个硬件定时器。这也就道出了为什么要软件定时器的缘由。软件定时器是通过编写程序的方式，将一个硬件定时器扩展出多个软件定时器，所扩展出的定时器数目可以根据系统的需要灵活定制。本章就 ClearRTOS 中的软件定时器进行详细介绍。

24.1 软件定时器分类

首先，软件定时器可分为一次性定时器和周期性定时器。对于一次性定时器，当定时器到期以后就不再有效了。周期性定时器则不同，只要没有被停止，定时器的回调函数就会根据设定的时间间隔周而复始地被调用。

在不少系统中，为了简化实现，并不直接提供周期性定时器的创建函数。在这种情形下，用户可以在一次性定时器的回调函数中，再一次以相同的时间间隔重新启动定时器，通过这种方式就可以达到与周期性定时器一样的效果，ClearRTOS 采用的也是这种方法。

其次，定时器根据到期时的回调函数调用环境的不同，可以分为中断回调定时器和任务回调定时器。中断回调定时器是指定时器的回调函数是在中断服务程序中被（间接）调用的，而任务回调定时器是指定时器的回调函数是由任务完成的（该任务是定时器管理模块的一部分）。

采用中断回调方式的好处是实时性较好，但这种方式也存在弊端。由于回调函数是在中断服务程序中被调用的，这就可能造成低优先级中断被延时响应，延时的长短与所到期定时器回调函数的实现复杂度有关。反之，采用任务回调就没有中断回调带来的中断延时问题，但其实时性却要差一点，为了提高实时性，可以考虑将回调任务的优先级调得高一点。

24.2 设计思路

当一个定时器被创建以后，系统怎样感知到这个定时器在什么时候到期呢？这需要依赖于

硬件定时器。硬件定时器周期性地产生中断，定时器管理模块在中断服务程序中检查定时器是否到期，这个硬件定时器就是"滴答"。由此看来，"滴答"的周期性将影响定时器的粒度和误差。比如，在一个"滴答"周期为 10 毫秒的操作系统中，定时器的粒度不可能小于 10 毫秒，而定时器的最大误差是 10 毫秒。

显然，当一个"滴答"中断发生时，考虑中断响应问题，我们不能在"滴答"的中断服务程序中将所有定时器遍历一遍以检查其是否到期，而应该设计一定的算法来减少每次所需检查的定时器数目，但同时也要考虑内存开销。

ClearRTOS 采用了"桶（bucket）"算法。现在假设用多个"桶"来装定时器。定时器按照一定的规则放入不同的"桶"中，当"滴答"产生中断时，只需检查当前的"桶"中是否有定时器，如果有就进行到期检查。"滴答"每到期一次，图 24.1 中的实线箭头就向右移动一个桶，我们称箭头正指向的"桶"为"当前桶"。

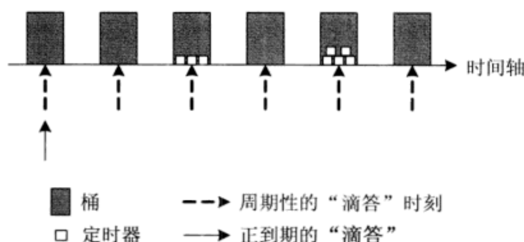


图 24.1

考虑内存开销问题，系统中"桶"的数量一定是有限的，也就是说，我们不可能为每个定时器设定一个桶。那定时器以什么规则放入桶中，即每个定时器对应的桶地址是什么呢？我们通过除留余数法得到。先为每个"桶"编上序号，以当前"桶"依次从 0 开始，接着将定时毫秒值转换为"滴答"数，然后用得到的"滴答"数与桶数量相除，得到的余数即为该定时器的桶地址。同时将相除得到的倍数值作为该定时器的属性保存起来，每当桶被检查一次时，该倍数值就减一，直到为 0 则说明该定时器到期。由此也可知，系统中的"桶"会被循环检查。

24.3 中断回调定时器

下面我们先介绍 ClearRTOS 内中断回调定时器的实现，后面在基于这一实现的基础上再探讨任务回调定时器的实现。

24.3.1 程序实现

图 24.2 列出了实现所需的基本数据结构。第 35 行定义了"滴答"的周期为 10 毫秒。第 37 行将 type_timer 结构的指针定义为 timer_handle_t。第 40 行定义了定时器到期回调函数的原型，其第一个参数是到期定时器，第二个参数是定时器在启动时所指定的参数。

```
00034: // in macrosecond
00035: #define CONFIG_TICK_DURATION_IN_MSEC 10
00036:
00037: typedef struct type_timer *timer_handle_t;
00038:
00039: // callback function for timer expiration
00040: typedef void (*expiry_callback_t)(timer_handle_t _handle, void *_arg);
00041:
00042: typedef enum {
00043:     TIMER_CREATED,
00044:     TIMER_STARTED,
00045:     TIMER_STOPPED
00046: } timer_state_t;
00047:
00048: typedef enum {
00049:     TIMER_TYPE_INTERRUPT,
00050:     TIMER_TYPE_TASK
00051: } timer_type_t;
00052:
00053: typedef struct type_timer {
00054:     dll_node_t node_;
00055:     expiry_callback_t callback_;
00056:     timer_state_t state_;
00057:     void *arg_;
00058:     usize_t ticks_;
00059:     usize_t bucket_index_;
00060:     usize_t round_;
00061:     magic_number_t magic_number_;
00062:     char name_ [NAME_MAX_LENGTH + 1];
00063: } timer_instance_t;
```

图 24.2

第 42～46 行定义了定时器状态，定时器的状态会因为函数调用而变迁，图 24.3 示例说明了其状态机。在后面讲解定时器的操作函数时，读者可以通过对照这一状态机以帮助理解。第 48～51 行定义了中断回调和任务回调两种定时器类型。

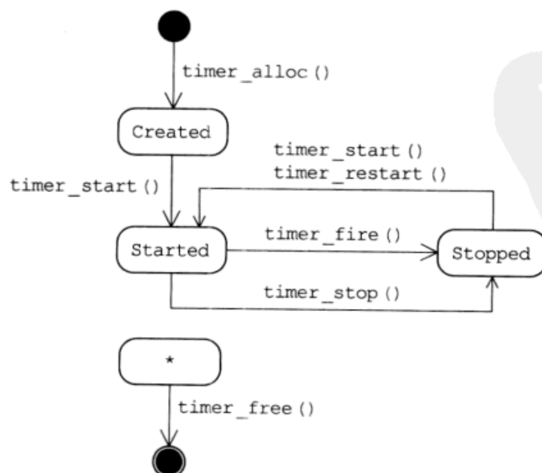


图 24.3

第 53～63 行定义了定时器的管理数据结构，其中各域的作用如下。

- node_：其类型是 dll_node_t，当定时器被放入“桶”中时需要通过这个数据结构连接到“桶”的链表中。
- callback_：保存的是定时器到期回调函数。
- state_：用于表示定时器所处状态。
- arg_：存放的是定时器到期回调函数的第二个参数，该参数由用户在启动定时器时指定。
- ticks_：记录的是定时器定时值，即定时器启动以后经过几个“滴答”后到期。
- bucket_index_：用于指示定时器是放在哪一个“桶”中的，即定时器对应的“桶”地址。系统中所有的“桶”以数组形式定义，该变量即为“桶”在数组内的索引。
- round_：用于表示一个定时器被循环检查多少圈后才到期。
- name_：用于保存定时器的名称。

图 24.4 列出了其他内容。

```
embedded/code/platform/timer/v1/src/timer.c
00035: #define CONFIG_MAX_BUCKET       13
00036: #define BUCKET_LAST_INDEX       (CONFIG_MAX_BUCKET - 1)
00037:
00038: #define CONFIG_MAX_TIMER        128
00039: #define TIMER_LAST_INDEX        (CONFIG_MAX_TIMER - 1)
00040:
00041: #define MAGIC_NUMBER_TIMER      0x54494D45L
00042:
00043: #define is_invalid_handle(_handle) \
00044:     ((_handle == null) || ((_handle)->magic_number_ != MAGIC_NUMBER_TIMER))
00045:
00046: typedef struct {
00047:     dll_t dll_;
00048:     statistic_t hit_;
00049: } bucket_t;
00050:
00051: typedef struct {
00052:     statistic_t notimer_;
00053:     statistic_t traversed_;
00054:     statistic_t abnormal_;
00055: } timer_statistic_t;
00056:
00057: static timer_instance_t g_timer_pool [CONFIG_MAX_TIMER];
00058: static dll_t g_free_timer;
00059: static dll_t g_inactive_timer;
00060:
00061: static usize_t g_cursor;
00062: static bucket_t g_buckets [CONFIG_MAX_BUCKET];
00063: static timer_statistic_t g_statistics;
00064: static timer_handle_t g_timer_next;
00065: static bucket_t *g_bucket_firing;
```

图 24.4

第 35 行定义了总的“桶”数，这里被定义成 13。关于系统中“桶”数量的确定，我们需要兼顾中断响应速度及系统存储空间。“桶”数量少，每个“桶”中的定时器数量会增多，即

“滴答”到期时需要一次性处理的定时器增多，这将影响“滴答”中断的响应速度。毫无疑问，“桶”数量越多需要开辟的内存空间就越大。

第 38 行定义了所支持的定时器个数。一个系统所需定时器的数量可根据下面将要讲到的统计信息进行调整。需要注意的是，由于 ClearRTOS 中每一个任务都需要创建一个定时器用于任务等待处理，因此所定义的定时器个数必须大于所需创建的任务个数。

第 41 行定义了定时器是否有效的标识值。第 43 行定义的宏用于有效性判断。

第 46～49 行定义了“桶”数据结构。其中的 dll_ 变量是一个链表，用于存放定时器；hit_ 变量用于统计“桶”中放入定时器的次数。

第 51～55 行定义的是用于存放统计信息的数据结构。notimer_ 变量中记录的是定时器不够用所出现的次数；traversed_ 变量中记录的是整个系统遍历“桶”的次数；abnormal_ 变量记录的是定时器模块出现错误的次数。

第 57 行定义了定时器数组，数组中的每一个元素代表可分配的定时器。定时器数组 g_timer_pool 中的各元素在没有被分配出去之前会被放入第 58 行定义的 g_free_timer 链表中，分配出去但是并没有被激活的定时器将会被放入第 59 行定义的 g_inactive_timer 链表中。而被激活的定时器是放在“桶”中的。

第 61 行定义的 g_cursor 变量前面已谈到，其指向 “当前桶”。第 62 行定义了“桶”数组。第 63 行对统计数据结构进行实例化。第 64 和 65 行定义的两个变量的作用留到后面再讨论。

24.3.1.1　分配定时器

一个定时器的获得需要通过调用 timer_alloc()函数，其实现如图 24.5 所示[①]。

```
embedded/code/platform/timer/v1/src/timer.c
00134: static void timer_init ()
00135: {
00136:     usize_t index;
00137:
00138:     for (index = 0; index <= TIMER_LAST_INDEX; ++ index) {
00139:         dll_push_tail (&g_free_timer, &g_timer_pool [index].node_);
00140:     }
00141: }
00142:
00143: error_t timer_alloc (timer_handle_t *_p_handle, const char *_name,
00144:     timer_type_t _type)
00145: {
00146:     interrupt_level_t level;
00147:     static bool initialized = false;
00148:     timer_handle_t handle;
00149:
```

① 原本想取名为“timer_create”，但这一名字已被 C 标准库所使用，为了避免冲突而选择了“timer_alloc”。

```
00150:     UNUSED (_type);
00151:
00152:     if (0 == _p_handle) {
00153:         return ERROR_T (ERROR_TIMER_ALLOC_INVHANDLE);
00154:     }
00155:
00156:     *_p_handle = null;
00157:     level = global_interrupt_disable ();
00158:     if (!initialized) {
00159:         timer_init ();
00160:         initialized = true;
00161:     }
00162:
00163:     handle = (timer_handle_t)dll_pop_head (&g_free_timer);
00164:     if (0 == handle) {
00165:         g_statistics.notimer_ ++;
00166:         global_interrupt_enable (level);
00167:         return ERROR_T (ERROR_TIMER_ALLOC_NOTIMER);
00168:     }
00169:     global_interrupt_enable (level);
00170:
00171:     handle->magic_number_ = MAGIC_NUMBER_TIMER;
00172:     handle->state_ = TIMER_CREATED;
00173:     dll_node_init (&handle->node_);
00174:     if (0 == _name) {
00175:         handle->name_ [0] = 0;
00176:     }
00177:     else {
00178:         strncpy (handle->name_, _name, (usize_t)sizeof (handle->name_));
00179:         handle->name_ [sizeof (handle->name_) - 1] = 0;
00180:     }
00181:
00182:     level = global_interrupt_disable ();
00183:     dll_push_tail (&g_inactive_timer, &handle->node_);
00184:     global_interrupt_enable (level);
00185:
00186:     *_p_handle = handle;
00187:     return 0;
00188: }
```

图 24.5

第 134～141 行的 timer_init()函数是模块的初始化函数。初始化过程就是将所有 g_timers 数组中的元素放入 g_free_timer 链表中。

timer_alloc()函数的第一个参数用于返回被创建定时器的句柄；第二个参数为定时器名称；第三个参数用于指定定时器的类型，该参数目前没有被使用。当第一次调用该函数时，会先调用 timer_init()函数进行定时器管理模块的初始化，对应代码在第 158～161 行。第 152 行检查参数_p_handle 是否为 null。第 163 行从空闲定时器链表中获取一个定时器实例，并在第 164～168 行检查是否有空闲定时器，如果没有则需要更新 notimer_统计变量。

第 157 行关闭的中断在第 169 行加以恢复，因为随后的初始化操作不存在竞争问题。第 171 行设置定时器的标识值。第 172 行设置定时器的状态为“已创建”。第 173 行对定时器的链表节点进行初始化。第 174～180 行初始化定时器名。第 183 行将定时器放入“未激活”链表中，

这一操作需要关闭中断以防止出现竞争问题。第 186 行将定时器返回给调用者。

24.3.1.2 释放定时器

定时器释放函数 timer_free()的实现如图 24.6 所示。

```
00190: error_t timer_free (timer_handle_t _handle)
00191: {
00192:     interrupt_level_t level;
00193:
00194:     level = global_interrupt_disable ();
00195:     if (is_invalid_handle (_handle)) {
00196:         global_interrupt_enable (level);
00197:         return ERROR_T (ERROR_TIMER_FREE_INVHANDLE);
00198:     }
00199:     if (TIMER_STARTED == _handle->state_) {
00200:         if (g_timer_next == _handle) {
00201:             g_timer_next = (timer_handle_t) dll_next (
00202:                 &g_bucket_firing->dll_, &_handle->node_);
00203:         }
00204:         dll_remove (&g_buckets [_handle->bucket_index_].dll_, &_handle->node_);
00205:     }
00206:     else {
00207:         dll_remove (&g_inactive_timer, &_handle->node_);
00208:     }
00209:     _handle->magic_number_ = 0;
00210:     dll_push_tail (&g_free_timer, &_handle->node_);
00211:     global_interrupt_enable (level);
00212:
00213:     return 0;
00214: }
```

图 24.6

第 195～198 行对定时器的有效性进行检查。第 199～208 行将定时器从相应的“桶”中移除。第 200～203 行对已启动的定时器需要进行额外的处理，其作用在讲解 timer_fire()函数时再补充。没有启动的定时器是放在“未激活”链表中的，第 207 行的用处就是将定时器从链表中删除。第 209 行通过设置定时器的标识值为 0 使其无效。第 210 行将被释放的定时器放回空闲链表中。

24.3.1.3 启动定时器

定时器进入启动状态是从调用 timer_start()函数开始的，图 24.7 是其程序实现。

```
00229: error_t timer_start (timer_handle_t _handle, msecond_t _duration,
00230:     expiry_callback_t _cb, void *_arg)
00231: {
00232:     interrupt_level_t level;
00233:
00234:     if (null == _cb) {
00235:         return ERROR_T (ERROR_TIMER_ALLOC_INVCB);
00236:     }
```

```
00237:     level = global_interrupt_disable ();
00238:     if (is_invalid_handle (_handle)) {
00239:        global_interrupt_enable (level);
00240:        return ERROR_T (ERROR_TIMER_START_INVHANDLE);
00241:     }
00242:     if (TIMER_STARTED == _handle->state_) {
00243:        g_statistics.abnormal_ ++;
00244:        global_interrupt_enable (level);
00245:        return ERROR_T (ERROR_TIMER_START_INVSTATE);
00246:     }
00247:
00248:     _handle->ticks_ = _duration / CONFIG_TICK_DURATION_IN_MSEC;
00249:     if (0 == _handle->ticks_) {
00250:        _handle->ticks_ ++;
00251:     }
00252:     _handle->callback_ = _cb;
00253:     _handle->arg_ = _arg;
00254:     timer_insert (_handle);
00255:     global_interrupt_enable (level);
00256:     return 0;
00257: }
```

图 24.7

timer_start()函数需要四个参数。第一个参数是定时器；第二个参数是定时器的定时毫秒值；第三个参数是定时器到期时的回调函数；第四个参数则是传递给回调函数的（第二个）参数。

第 234～241 行进行输入参数有效性检查。从图 24.3 可以看出，一个能被启动的定时器的状态只能是“已创建”，第 242～246 行对状态加以确认。如果不是则更新 abnormal_统计变量并报错。第 248 行计算定时器到期所应等待的“滴答”次数。第 249～251 行判断如果“滴答”值为 0，则置为 1。第 254 行调用 timer_insert()函数将被启动的定时器放入相应的“桶”中。timer_insert()函数的实现如图 24.8 所示。

```
embedded/code/platform/timer/v1/src/timer.c
00216: static void timer_insert (timer_handle_t _handle)
00217: {
00218:     _handle->bucket_index_ = (g_cursor + _handle->ticks_) % CONFIG_MAX_BUCKET;
00219:     _handle->round_ = _handle->ticks_ / CONFIG_MAX_BUCKET;
00220:     if (g_bucket_firing == &g_buckets [_handle->bucket_index_]) {
00221:        _handle->round_ --;
00222:     }
00223:     dll_remove (&g_inactive_timer, &_handle->node_);
00224:     dll_push_head (&g_buckets [_handle->bucket_index_].dll_, &_handle->node_);
00225:     g_buckets [_handle->bucket_index_].hit_ ++;
00226:     _handle->state_ = TIMER_STARTED;
00227: }
```

图 24.8

第 218 行的作用是计算定时器应放入哪一个“桶”中。第 219 行计算“滴答”要绕着所有“桶”“跑”多少圈定时器才到期。第 220～222 行是针对定时器所需放入的“桶”刚好是“滴答”中断正在处理的那个。在这种情形下，需要对 round_变量进行减一操作，因为第 224 行是将定时器放入“桶”的头部而使得跳过一次处理机会。第 223 行将定时器从“未激活”链表中

删除。第 225 行更新“桶”被使用的统计计数。在第 226 行将定时器的状态设置为“已启动”。

在本章的开头指出，ClearRTOS 并不直接支持周期性定时器，而是需要通过一次性定时器来实现。为了简化用户操作，ClearRTOS 提供了 timer_restart()函数。在一次性定时器的回调函数中，通过调用 timer_restart()函数可以方便地实现周期性定时器。timer_restart()函数的实现如图 24.9 所示。

```
embedded/code/platform/timer/v1/src/timer.c
00259: error_t timer_restart (timer_handle_t _handle)
00260: {
00261:     interrupt_level_t level;
00262:
00263:     level = global_interrupt_disable ();
00264:     if (is_invalid_handle (_handle)) {
00265:         global_interrupt_enable (level);
00266:         return ERROR_T (ERROR_TIMER_RESTART_INVHANDLE);
00267:     }
00268:     if (TIMER_STOPPED != _handle->state_) {
00269:         g_statistics.abnormal_ ++;
00270:         global_interrupt_enable (level);
00271:         return ERROR_T (ERROR_TIMER_RESTART_INVSTATE);
00272:     }
00273:
00274:     timer_insert (_handle);
00275:     global_interrupt_enable (level);
00276:     return 0;
00277: }
```

图 24.9

timer_restart()函数的实现比 timer_start()函数还要简单，因为它是基于定时器已经使用 timer_start()函数启动过这一前提假设的，这也是为什么在第 268 行存在判断定时器是否处于“已停止”这一状态的原因。只要定时器被启动过了，它的 ticks_、callback_和 arg_变量都已被初始化过了，直接使用就行了。

24.3.1.4　停止定时器

如果希望停止一个已被启动且没有到期的定时器，需要调用 timer_stop()函数，其实现可以从图 24.10 中找到。

```
embedded/code/platform/timer/v1/src/timer.c
00279: error_t timer_stop (timer_handle_t _handle)
00280: {
00281:     interrupt_level_t level;
00282:
00283:     level = global_interrupt_disable ();
00284:     if (is_invalid_handle (_handle)) {
00285:         global_interrupt_enable (level);
00286:         return ERROR_T (ERROR_TIMER_STOP_INVHANDLE);
00287:     }
00288:     if (_handle->state_ != TIMER_STARTED) {
00289:         g_statistics.abnormal_ ++;
00290:         global_interrupt_enable (level);
```

```
00291:         return ERROR_T (ERROR_TIMER_STOP_INVSTATE);
00292:     }
00293:     if (g_timer_next == _handle) {
00294:         g_timer_next = (timer_handle_t) dll_next
00295:             (&g_bucket_firing->dll_, &_handle->node_);
00296:     }
00297:     _handle->state_ = TIMER_STOPPED;
00298:     dll_remove (&g_buckets [_handle->bucket_index_].dll_, &_handle->node_);
00299:     dll_push_tail (&g_inactive_timer, &_handle->node_);
00300:     global_interrupt_enable (level);
00301:
00302:     return 0;
00303: }
```

图 24.10

第 284～292 行用于确保参数的有效性和检查定时器的状态。g_timer_next 变量中保存的是 timer_fire()将要处理的下一个定时器。第 293～296 行如果判断到被停止的定时器刚好是该定时器，就更新 g_timer_next 变量指向后一个，使得 timer_fire()函数跳过对其处理。

第 297 行将定时器的状态设置为停止。第 298 行把定时器从“桶”中移除，并在第 299 行将其放入“未激活”链表中。

24.3.1.5 定时器到期处理

当一个“滴答”中断发生时，timer_fire()函数会被调用，以处理 “当前桶”中的定时器，其实现如图 24.11 所示[②]。

```
00067: void timer_fire ()
00068: {
00069:     interrupt_level_t level;
00070:     timer_handle_t handle;
00071:
00072:     level = global_interrupt_disable ();
00073:     g_bucket_firing = &g_buckets [g_cursor];
00074:     if (0 == dll_size (&g_bucket_firing->dll_)) {
00075:         // no timer is expired
00076:         goto out;
00077:     }
00078:
00079:     handle = (timer_handle_t) dll_head (&g_bucket_firing->dll_);
00080:     while (0 != handle) {
00081:         g_statistics.traversed_ ++;
00082:         g_timer_next = (timer_handle_t) dll_next (
00083:             &g_bucket_firing->dll_, &handle->node_);
00084:         if (handle->round_ > 0) {
00085:             // in this case the timer is still not expired
00086:             handle->round_ --;
00087:         }
```

② 在模拟环境运行的 ClearRTOS 中，timer_fire()函数是由空闲任务调用的而不是真正的“滴答”中断服务程序。别忘了，ClearRTOS 无法接管 Linux 操作系统上的中断。

```
00088:         else {
00089:             // hooray, the timer is expired
00090:             dll_remove (&g_bucket_firing->dll_, &handle->node_);
00091:             dll_push_tail (&g_inactive_timer, &handle->node_);
00092:             handle->state_ = TIMER_STOPPED;
00093:             global_interrupt_enable (level);
00094:             handle->callback_(handle, handle->arg_);
00095:             level = global_interrupt_disable ();
00096:         }
00097:         handle = g_timer_next;
00098:     }
00099:
00100: out:
00101:     g_cursor ++;
00102:     if (g_cursor > BUCKET_LAST_INDEX) {
00103:         g_cursor = 0;
00104:     }
00105:     g_timer_next = 0;
00106:     g_bucket_firing = 0;
00107:     global_interrupt_enable (level);;
00108: }
```

图 24.11

第 73 行初始化 g_dll_firing 全局变量（定义于图 24.4 的 65 行）以指向正在处理的“桶”。第 74～77 行查看“桶”中是否有定时器，如果没有则直接跳转到 out 标签处。程序运行到第 79 行说明被处理的“桶”中存在定时器。第 80～98 行的 while()语句块将依次检查“桶”中的定时器是否到期。

第 81 行更新统计信息表示对桶进行过一次遍历。第 82 行让 g_timer_next 变量指向下一个要被处理的定时器。第 84～87 行针对的是定时器的 round_变量不为 0 的情形，这种情形说明定时器并没有到期，因此只需对 round_变量进行减一操作。第 88～96 行是针对定时器到期的情形。第 97 行将 g_timer_next 变量的值赋给 handle 变量。在前面的 timer_stop()函数中，我们已经看到 g_timer_next 变量可能因为停止定时器而更新。

当一个定时器到期时，在第 90～92 行先将其从“桶”中移除并放入“未激活”链表中，且设置定时器的状态为“已停止”。第 94 行调用定时器的回调函数。在调用定时器回调函数的前后，分别存在恢复和关闭中断的操作。注意在第 94 行以后，不能再使用 handle 所指向的定时器，因为该定时器有可能在中断服务程序中被删除。

关于 g_next_firing 和 g_dll_firing 两个全局变量需要详细讲解一下。引入这两个全局变量正是因为在调用定时器的回调函数之前在第 93 行存在恢复中断这一操作。我们不能忽视这一小步的中断恢复操作，其所引入的竞争问题很容易被忽视。图 24.12 示例说明了 timer_fire()函数中的所有中断控制点以帮助分析问题。

对照代码读者将发现，在实际的代码中，p3 点对 g_cursor 变量的操作还包含回转处理，图中为了简化只列出了 g_cursor++。请注意，图中 p1 到 p2 是调用定时器回调函数的时期，由于 p1 点的中断恢复操作将造成在这一期间可以发生很多的事情。比如，有可能发生比“滴答”中断优

先级更高的中断，且在该中断的服务程序中需要释放定时器。当所删除的定时器刚好位于 timer_fire()函数正在处理的“桶”中时就可能影响 timer_fire()函数的行为。如果所释放的定时器刚好是 g_next_firing 所指向的话，timer_fire()函数就不应对其进行到期处理，这就是为什么在图 24.6 中存在第 200～203 行代码的缘由。相类似的代码同样存在于图 24.10 中的第 293～296 行。

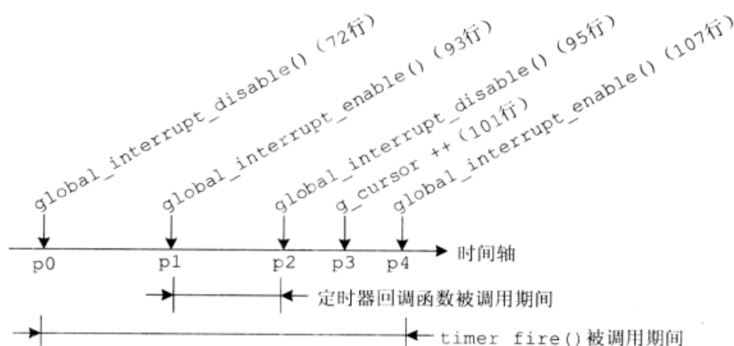


图 24.12

如果在 p1 至 p2 期间刚好需要启动定时器，且这个定时器又需要放到 g_dll_firing 所指向的“桶”中，则会发生定时器的定时值被放大的情形。举例来说，当 CONFIG_MAX_BUCKETS 为 13，“滴答”周期为 10 毫秒的情形下，如果在 p1 至 p2 期间插入了一个 130 毫秒的定时器的话，这个定时器刚好要放入到 g_dll_firing 所指向的链表中，且 round_值将为 1。由于在 timer_start()函数中是将新建的定时器放入 g_dll_firing 的最前面的，此时 round_值并不会在 timer_fire()函数的这次调用期间减一，这导致定时器的实际定时值变为 260 毫秒。为了解决这一问题，需要在图 24.8 中的第 221 行对 round_变量的值先减一。

24.3.1.6　模块管理

定时器模块只关注系统终止化时是否回收了所有的定时器，模块回调函数 module_timer() 的实现如图 24.13 所示。

```
00110: static bool timer_check_for_each (dll_t *_p_dll, dll_node_t *_p_node, void *_p_arg)
00111: {
00112:     timer_handle_t handle = (timer_handle_t)_p_node;
00113:
00114:     UNUSED (_p_dll);
00115:     UNUSED (_p_arg);
00116:
00117:     console_print ("Error: timer \"%s\" isn't deleted\n", handle->name_);
00118:     return true;
00119: }
00120:
00121: error_t module_timer (system_state_t _state)
00122: {
00123:     usize_t idx;
00124:
```

```
00125:    if (STATE_DESTROYING == _state) {
00126:        for (idx = 0; idx <= BUCKET_LAST_INDEX; ++ idx) {
00127:            (void) dll_traverse (&g_buckets [idx].dll_, timer_check_for_each, 0);
00128:        }
00129:        (void) dll_traverse (&g_inactive_timer, timer_check_for_each, 0);
00130:    }
00131:    return 0;
00132: }
```

图 24.13

在检查定时器是否完全回收时，第 127～129 行不仅检查了各个“桶”，还检查了“未激活”链表。对于没有回收的定时器，会在 timer_check_for_each()函数中以错误日志的形式报告。

通过调用 timer_dump()函数可以显示定时器模块的相关信息，其实现如图 24.14 所示。

```
00322: void timer_dump ()
00323: {
00324:    usize_t index;
00325:
00326:    scheduler_lock ();
00327:    console_print ("\n");
00328:    console_print ("Summary\n");
00329:    console_print ("-------\n");
00330:    console_print (" Supported: %u\n", CONFIG_MAX_TIMER);
00331:    console_print (" Allocated: %u\n", CONFIG_MAX_TIMER -
00332:    dll_size (&g_free_timer));
00333:    console_print (" .BSS Used: %u\n", ((address_t)&g_bucket_firing
00334:        - (address_t)g_timer_pool) + sizeof (g_bucket_firing));
00335:    console_print ("\n");
00336:    console_print ("Statistics\n");
00337:    console_print ("----------\n");
00338:    console_print (" No Timer: %u\n", g_statistics.notimer_);
00339:    console_print (" Abnormal: %u\n", g_statistics.abnormal_);
00340:    console_print (" Traversed: %u\n", g_statistics.traversed_);
00341:    console_print ("\n");
00342:    console_print ("Bucket Details\n");
00343:    console_print ("--------------\n");
00344:    for (index = 0; index <= BUCKET_LAST_INDEX; ++ index) {
00345:        console_print (" [%u]: hit (%u), timer held (%u)\n", index,
00346:            g_buckets [index].hit_, dll_size (&g_buckets [index].dll_));
00347:    }
00348:    console_print ("\n");
00349:    scheduler_unlock ();
00350: }
```

图 24.14

24.3.2　timerv1 示例程序

图 24.15 所示的代码可用于验证中断回调定时器模块的功能。

```
00026: #include "main.h"
00027: #include "device.h"
00028: #include "console.h"
```

```
00029: #include "timer.h"
00030:
00031: #define DURATION_ONE_SECOND    1000
00032:
00033: static void log_timer_event (const char *_name)
00034: {
00035:     if (is_in_interrupt ()) {
00036:         console_print ("Info: timer \"%s\" is expired [in interrupt]\n", _name);
00037:     }
00038:     else {
00039:         console_print ("Info: timer \"%s\" is expired [in \"%s\" task]\n", _name,
00040:             task_self ()->name_);
00041:     }
00042: }
00043:
00044: static void callback_timer (timer_handle_t _handle, void *_arg)
00045: {
00048:     UNUSED (_arg);
00049:
00050:     (void) timer_restart (_handle);
00051:     log_timer_event (_handle->name_);
00052: }
00053:
00054: static void task_test (const char _name [], void *_p_arg)
00055: {
00056:     timer_handle_t handle = (timer_handle_t)_p_arg;
00057:
00058:     UNUSED (_name);
00059:
00060:     (void) timer_start (handle, DURATION_ONE_SECOND, callback_timer, 0);
00061:     (void) task_sleep (11000);
00062:     timer_dump ();
00063:     multitasking_stop ();
00064: }
00065:
00066: error_t module_testapp (system_state_t _state)
00067: {
00068:     static device_handle_t ctrlc_handle;
00069:     static task_handle_t handle;
00070:     STACK_DECLARE (stack, 2048);
00071:     static timer_handle_t timer;
00072:
00073:     if (STATE_INITIALIZING == _state) {
00074:         (void) timer_alloc (&timer, "One Second", TIMER_TYPE_INTERRUPT);
00075:
00076:         (void) task_create (&handle, "Test", 16, stack, sizeof (stack));
00077:         (void) task_start (handle, task_test, timer);
00078:
00079:         (void) device_open (&ctrlc_handle, "/dev/ui/ctrlc", 0);
00080:     }
00081:     else if (STATE_DESTROYING == _state) {
00082:         (void) device_close (ctrlc_handle);
00083:         (void) task_delete (handle);
00084:         (void) timer_free (timer);
00085:     }
00086:     return 0;
00087: }
00088:
00089: int module_registration_entry (int argc, char *argv [])
00090: {
```

```
00091:     UNUSED (argc);
00092:     UNUSED (argv);
00093:
00094:     (void) module_register ("Interrupt", MODULE_INTERRUPT, CPU_LEVEL,
               module_interrupt);
00095:     (void) module_register ("Device", MODULE_DEVICE, DRIVER_LEVEL, module_device);
00096:     (void) module_register ("Timer", MODULE_TIMER, OS_LEVEL, module_timer);
00097:     (void) module_register ("Task", MODULE_TASK, OS_LEVEL, module_task);
00098:     (void) module_register ("TestApp", MODULE_TESTAPP, APPLICATION_LEVEL3,
               module_testapp);
00099:     return 0;
00100: }
```

图 24.15

第 31 行定义了要创建定时器的定时值，在此定义为 1000 毫秒。在 module_testapp()函数的第 74 行创建一个名为“One Second”的定时器，第 76～77 行创建并启动用于测试的“Test”任务。第 60 行在“Test”任务体函数中启动了定时器。

定时器的到期回调函数是 callback_timer()，其实现位于第 44～52 行。其中第 50 行调用 timer_restart()函数使得“One Second”定时器成为一个周期性定时器；第 51 行调用 log_timer_event()函数将定时器的到期事件打印出来。

第 33～42 行的 log_timer_event()函数，将定时器的名称及其是处于中断状态被调用还是任务状态被调用这一信息打印出来。示例程序的运行结果列于图 24.16 中。

```
                    /embedded/build
$ ./release/timerv1.exe

System is going to be up

Info: press Ctrl-C to terminate!
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]

Summary
-------
  Supported: 128
  Allocated: 3
  .BSS Used: 6948

Statistics
----------
   No Timer: 0
   Abnormal: 0
  Traversed: 173

Bucket Details
```



图 24.16

从运行结果来看，“One Second”定时器属于中断回调定时器。从统计信息来看，各个“桶”的使用也比较均匀，这说明“桶”的个数比较合理。

24.4 定时误差

实现中有两处会带来定时误差，我们在实际应用中需要充分考虑这些误差。第一处，当定时值不是“滴答”的倍数时，把定时值转换为“滴答”数会造成误差。

第二处，timer_start()被调用的时刻与“滴答”中断的发生时刻不同步而导致的误差。图 24.17 示例说明了在 t0 和 t1 时刻分别启动两个 10 毫秒定时器的情形，图中假设“滴答”的周期也是 10 毫秒。对于 t0 时刻启动的定时器，其误差是图中的 e0，而 t1 时刻启动的定时器其误差是 e1，两种情形下定时器的实际定时值比所设置的要小。

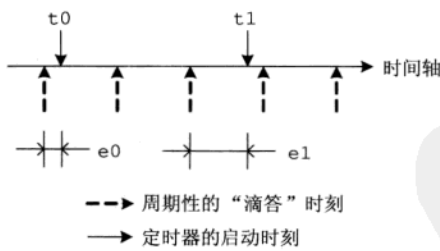


图 24.17

24.5 提高遍历效率

先假设某“桶”内的定时器分布如图 24.18 所示，请注意其中各定时器 round_变量的值。虽然此时只有 round_值为 0 的 t0 需要进行到期处理，但是，timer_fire()函数还是依次遍历“桶”

中的每一个定时器，在此需要遍历 4 次。我们可以考虑通过算法改进来提高遍历效率，这需要对“桶”中的定时器进行排序，以及改变 round_变量的含义。

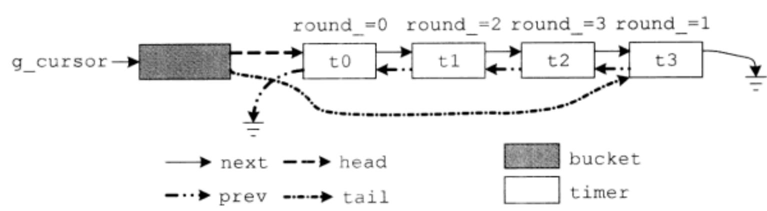


图 24.18

算法改进的第一步，是先对图 24.18 中的定时器按到期的时间进行排序，得到图 24.19 所示的链表结构。

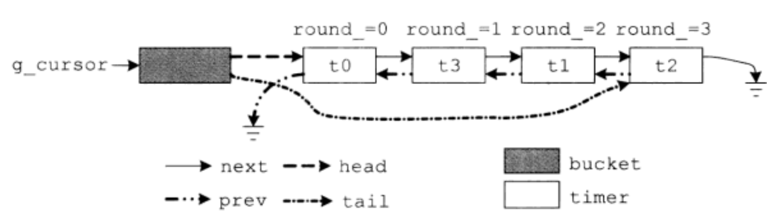


图 24.19

接着将 round_值从绝对值变为两个定时器之间的相对值，更改后的链表如图 24.20 所示。对于链表的第一个节点，因为它的前面并没有定时器，所以其 round_所指示的值仍是绝对值。

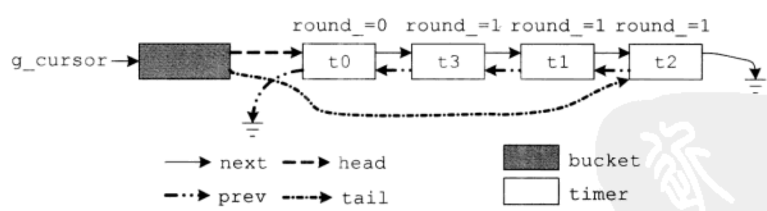


图 24.20

有了这些改动后，timer_fire()函数在遍历“桶”中的定时器时就不需要每次都遍历所有的定时器了，而是在碰到定时器 round_变量值不为 0 时对其减一后就终止。这是图 24.21 中所增加第 87 行的作用。

```
00067: void timer_fire ()
00068: {
00069:     ......
00080:     while (0 != handle) {
00081:         g_statistics.traversed_ ++;
```

```
00082:         g_timer_next = (timer_handle_t) dll_next (
00083:             &g_bucket_firing->dll_, &handle->node_);
00084:         if (handle->round_ > 0) {
00085:             // in this case the timer is still not expired
00086:             handle->round_ --;
00087:             break;
00088:         }
00089:         ......
00109: }
```

图 24.21

由于 round_变量的含义发生了变化，当释放一个定时器时也需要对 round_变量进行更新，更改如图 24.22 所示。其中的第 207～211 行是新增的，即当被删除的定时器后面存在节点时，需要将被删除定时器的 round_值加到紧随其后的定时器上。

```
00191: error_t timer_free (timer_handle_t _handle)
00192: {
00193:     ......
00200:     if (TIMER_STARTED == _handle->state_) {
00201:         timer_handle_t next;
00202:
00203:         if (g_timer_next == _handle) {
00204:             g_timer_next = (timer_handle_t) dll_next (
00205:                 &g_bucket_firing->dll_, &_handle->node_);
00206:         }
00207:         next = (timer_handle_t)dll_next (&g_buckets [
00208:             _handle->bucket_index_].dll_, &_handle->node_);
00209:         if (0 != next) {
00210:             next->round_ += _handle->round_;
00211:         }
00212:         dll_remove (&g_buckets [_handle->bucket_index_].dll_, &_handle->node_);
00213:     }
00214:     ......
00222: }
```

图 24.22

同样的改动存在于 timer_stop()函数内，见图 24.23 中的第 334～338 行。

```
00314: error_t timer_stop (timer_handle_t _handle)
00315: {
00316:     interrupt_level_t level;
00317:     timer_handle_t next;
00318:
00319:     ......
00333:     _handle->state_ = TIMER_STOPPED;
00334:     next = (timer_handle_t)dll_next (&g_buckets [
00335:         _handle->bucket_index_].dll_, &_handle->node_);
00336:     if (0 != next) {
00337:         next->round_ += _handle->round_;
00338:     }
00339:     dll_remove (&g_buckets [_handle->bucket_index_].dll_, &_handle->node_);
00340:     dll_push_tail (&g_inactive_timer, &_handle->node_);
```

```
00341:     global_interrupt_enable (level);
00342:
00343:     return 0;
00344: }
```

图 24.23

新算法在启动一个定时器时需要在“桶”中找到合适的插入点，为此，需要增加一个新的函数——timer_dll_insert()，该函数被 timer_insert()函数调用，更改可以从图 24.24 中找到。

```
00242: static void timer_insert (timer_handle_t _handle)
00243: {
00244:     _handle->bucket_index_ = (g_cursor + _handle->ticks_) % CONFIG_MAX_BUCKET;
00245:     dll_remove (&g_inactive_timer, &_handle->node_);
00246:     _handle->round_ = _handle->ticks_ / CONFIG_MAX_BUCKET;
00247:     if (g_bucket_firing == &g_buckets [_handle->bucket_index_]) {
00248:         if(0 == g_timer_next || _handle->round_ <= g_timer_next->round_) {
00249:             g_timer_next = _handle;
00250:         }
00251:     }
00252:     if (0 == dll_size (&g_buckets [_handle->bucket_index_].dll_)) {
00253:         dll_push_tail (&g_buckets [_handle->bucket_index_].dll_,
00254:             &_handle->node_);
00255:     }
00256:     else {
00257:         (void)dll_traverse (&g_buckets [_handle->bucket_index_].dll_,
00258:             timer_dll_insert, (void *)&_handle->node_);
00259:     }
00260:     g_buckets [_handle->bucket_index_].hit_ ++;
00261:     _handle->state_ = TIMER_STARTED;
00262: }
```

图 24.24

第 245 行先将需要被启动的定时器从“未激活”链表中删除。第 247 行判断被启动的定时器对应的“桶”是否是 timer_fire()函数正在处理的。第 248 行判断如果“桶”中已没有接下来需要处理的定时器，或者被插入定时器的到期时间比 g_timer_next 所指向的更小，那么需要更新 g_timer_next 到该定时器。第 252～259 行的功能是将定时器插入“桶”中正确的位置。如果被插入的“桶”中没有定时器，在第 253～254 行直接将定时器放到链表中；否则运行第 257～258 行遍历链表找到正确的插入点。在遍历链表时，使用的回调函数是 timer_dll_insert()，这个函数的实现如图 24.25 所示。

```
00224: static bool timer_dll_insert (dll_t *_p_dll, dll_node_t *_p_node, void *_p_inserted)
00225: {
00226:     timer_handle_t inserting = (timer_handle_t) _p_node;
00227:     timer_handle_t inserted = (timer_handle_t) _p_inserted;
00228:
00229:     if (inserted->round_ <= inserting->round_) {
00230:         inserting->round_ -= inserted->round_;
00231:         dll_insert_before (_p_dll, &inserting->node_, &inserted->node_);
00232:         return false;
00233:     }
```

```
00234:     inserted->round_ -= inserting->round_;
00235:     if (0 == dll_next (_p_dll, _p_node)) {
00236:         dll_insert_after (_p_dll, &inserting->node_, &inserted->node_);
00237:         return false;
00238:     }
00239:     return true;
00240: }
```

图 24.25

timer_dll_insert()函数的第一个参数就是链表指针，第二个参数是当前正在遍历的节点，第三个参数来自于 dll_traverse()的最后一个参数，即需要被插入的节点。在第 229～233 行检查所插入节点的 round_变量值是否小于或等于正在遍历的节点。如果是，则说明需要插入到正在遍历节点的前面。在插入之前，在第 230 行需要从遍历节点的 round_变量中减去被插入节点 round_变量的值，然后在第 231 行调用 dll_insert_before()函数完成插入操作，并返回 false 以指示 dll_traverse()可以终止遍历动作了。

程序如果运行到第 234 行，则说明需要插入的节点应当放在当前遍历节点之后，因此需要从被插入节点中减去当前正在遍历节点 round_变量的值。第 235 行用于检查当前正被遍历的节点之后是否还有节点，如果没有则在第 236 行将被插入的节点插入并返回 false 表示没有必要进行后续的遍历了。

有了上面的这些更改后，对图 24.18 所示布局的定时器，第一次“滴答”到期时需要遍历的次数从最原始算法的 4 次减为 2 次，第一次“滴答”到期处理完后的链表结构将如图 24.26 所示。相似地，后面的遍历次数都将有所下降。

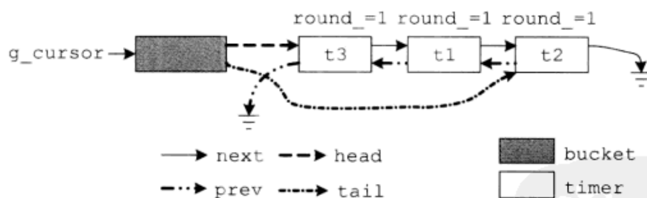


图 24.26

有读者可能会有疑问，觉得这一改进是将所花费的时间从 timer_fire()函数移到了其他函数中而已，实际上并没有提高效率。其实不然，因为启动和停止定时器的频率通常要明显地低于 timer_fire()函数的被调用频率，比如当“滴答”为 10 毫秒时，每秒钟要调用 timer_fire()函数 100 次。这对于中断回调定时器来说提高遍历效率就显得尤为重要了。

通过 timerv2 示例程序，可以帮助我们检查定时模块的这一改进是否有效。timerv2 示例程序的源代码与 timerv1 是完全一样的，唯一的区别是 timerv2 示例程序是与 libtimerv2.a 库进行链接而生成的。图 24.27 示例说明了改进之后 timerv2 示例程序的运行结果。通过与 timerv1 示例程序的运行结果比较，读者将发现“Traversed”统计信息从 173 下降到了 166。算法的改进

效果会随被创建定时器数量和定时值的增加而提高。

```
                    /embedded/build
$ ./release/timerv2.exe

System is going to be up

Info: press Ctrl+C to terminate!
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]
Info: timer "One Second" is expired [in interrupt]

Summary
-------
 Supported: 128
 Allocated: 3
 .BSS Used: 6948

Statistics
----------
  No Timer: 0
  Abnormal: 0
 Traversed: 166

Bucket Details
--------------
  [0]: hit (0), timer held (0)
  [1]: hit (1), timer held (0)
  [2]: hit (1), timer held (0)
  [3]: hit (1), timer held (0)
  [4]: hit (1), timer held (1)
  [5]: hit (1), timer held (0)
  [6]: hit (1), timer held (0)
  [7]: hit (1), timer held (0)
  [8]: hit (2), timer held (0)
  [9]: hit (1), timer held (0)
  [10]: hit (1), timer held (0)
  [11]: hit (1), timer held (0)
  [12]: hit (1), timer held (0)

System is going to be down
```

图 24.27

24.6　改善实时性

不恰当的中断控制时机会影响中断响应的实时性并带来更大的中断响应延时，图 24.28 示例说明了中断响应延时是如何发生的。

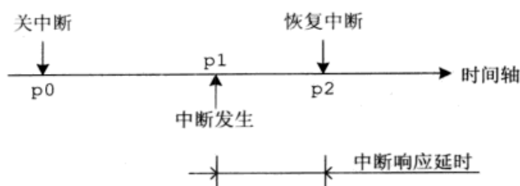


图 24.28

某函数因为预防竞争问题的需要在 p0 点关闭中断，然后在 p2 点恢复中断。如果在 p0 和 p2 点之间的 p1 点发生了中断，就会因为中断的暂时关闭而使得中断无法被处理器响应，而会延迟到 p2 点中断被恢复时才有可能被处理。图中的 p1 到 p2 点就是中断响应延时。中断响应延时除了影响中断外，还会间接影响任务响应的实时性。

同理，互斥锁的不合理使用会影响任务响应的实时性并带来更大的任务响应延时。任务响应延时可以通过图 24.29 加以理解。请注意，任务延时只会发生在任务间共用互斥锁这种情形，其不会影响中断的实时性。还有，互斥锁的优先级继承功能并不能减小图中所示的任务响应延时。

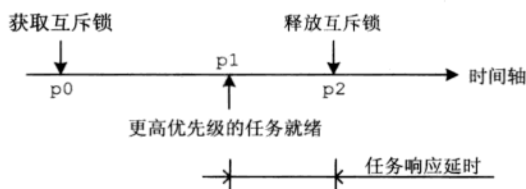


图 24.29

不论是中断响应延时还是任务响应延时要完全消除是不可能的，但好的设计能减小这个延时。要减小响应延时就必须尽可能减小上面两图中 p0 至 p2 的时间，这也是为什么在 27.1.12 节提出“青睐小粒度锁”这一编程好习惯的原因。本节我们以定时器的实现为例，看一看如何通过设计来改善其实时性。

24.6.1 实时性分析

定时器模块的实现存在两个比较耗时的地方，一个位于 timer_fire()函数内，另一个存在于 timer_start()函数中。这两个函数的共同点是存在对链表的遍历。

timer_fire()函数的实现中最为耗时的操作是回调函数的调用。但现有实现在每次调用回调函数之前都会先恢复中断，以及调用完了后又重新关闭中断，这个设计就是为了提高实时性。由于调用回调函数时并没有关闭中断，期间如果有中断发生处理器就有可能立即响应。timer_fire()函数的实现充分地考虑了实时性问题。

timer_start()函数通过调用 timer_insert()函数将被启动的定时器放入“桶”中，在确定定时

器的插入位置时需要通过遍历链表找到合适的点。如果链表较长所需遍历的定时器较多时就会造成中断的关闭时间加长。很明显，timer_start()函数的实现存在实时性改进空间。

24.6.2　改进实时性

提高 timer_start()函数实时性最基本的想法是：timer_start()函数在遍历链表的过程中进行多次的中断控制操作。

改进将从数据结构的变化开始，图 24.30 示例说明了对 bucket_t 数据结构的变更，以及新定义的宏——CONFIG_INTERRUPT_FLASH_FREQUENCY。

```
00046: #define CONFIG_INTERRUPT_FLASH_FREQUENCY     16
00047:
00051: typedef struct {
00052:     dll_t dll_;
00053:     statistic_t hit_;
00054:     statistic_t redo_;
00055:     usize_t reentrance_;
00056:     usize_t level_;
00057: } bucket_t;
```

图 24.30

第 46 行的 CONFIG_INTERRUPT_FLASH_FREQUENCY 表示当 timer_start()函数在遍历链表时每遍历多少个就需要恢复一次中断，在这里它的值被定义为 16。第 54 行定义了一个新的统计变量 redo_，以便记录 timer_start()函数在插入定时器到“桶”内的过程中一共进行了多少次重新遍历，具体的含义后面将看到。第 55 行的 reentrance_变量用于表示 timer_start()函数在某一时刻被重入的次数，它可以理解为一个计数器，记录了“同时”有几个任务或中断正在存取同一个“桶”。第 56 行的 level_变量表示在 timer_start()函数被调用的过程中，有多少次与“桶”相关的定时器进行过插入或删除操作。reentrance_和 level_变量的具体含义需要后面结合程序实现加深理解。

由于 timer_start()函数将调用 timer_insert()函数实现定时器的插入操作，因此 timer_start()函数的真正更改是体现在 timer_insert()函数内的，图 24.31 是 timer_insert()函数的新实现。它与前一版本的实现在很多地方是相似的，下面重点介绍不同点。

```
00305: static void timer_insert (timer_handle_t _handle)
00306: {
00307:     interrupt_level_t interrupt_level;
00308:     bucket_t *p_bucket;
00309:     usize_t count = 0, round, level;
00310:     timer_handle_t iterator;
00311:
00312:     interrupt_level = global_interrupt_disable ();
00313:     _handle->bucket_index_ = (g_cursor + _handle->ticks_) % CONFIG_MAX_BUCKET;
00314:     dll_remove (&g_inactive_timer, &_handle->node_);
```

```
00315:
00316:     p_bucket = &g_buckets [_handle->bucket_index_];
00317:     round = _handle->ticks_ / CONFIG_MAX_BUCKET;
00318:     level = ++ p_bucket->level_;
00319:     p_bucket->reentrance_ ++;
00320:
00321: redo:
00322:     iterator = (timer_handle_t) dll_head (&p_bucket->dll_);
00323:     _handle->round_ = round;
00324:     for (;;) {
00325:         if (0 == iterator) {
00326:             dll_push_tail (&p_bucket->dll_, &_handle->node_);
00327:             break;
00328:         }
00329:         if (_handle->round_ <= iterator->round_) {
00330:             iterator->round_ -= _handle->round_;
00331:            dll_insert_before (&p_bucket->dll_, &iterator->node_, &_handle->node_);
00332:             break;
00333:         }
00334:         _handle->round_ -= iterator->round_;
00335:         iterator = (timer_handle_t) dll_next (&p_bucket->dll_, &iterator->node_);
00336:
00337:         count ++;
00338:         if (count < CONFIG_INTERRUPT_FLASH_FREQUENCY) {
00339:             continue;
00340:         }
00341:
00342:         count = 0;
00343:         global_interrupt_enable (interrupt_level);
00344:         // at this moment we give a chance to the higher pirority
00345:         // interrupt (or task) for being served (or running)
00346:         interrupt_level = global_interrupt_disable ();
00347:         if (p_bucket->level_ != level) {
00348:             level = ++ p_bucket->level_;
00349:             p_bucket->redo_ ++;
00350:             goto redo;
00351:         }
00352:     }
00353:     if (g_bucket_firing == &g_buckets [_handle->bucket_index_]) {
00354:         if(0 == g_timer_next || _handle->round_ <= g_timer_next->round_) {
00355:             g_timer_next = _handle;
00356:         }
00357:     }
00358:     g_buckets [_handle->bucket_index_].hit_ ++;
00359:     if (0 == -- p_bucket->reentrance_) {
00360:         p_bucket->level_ = 0;
00361:     }
00362:     _handle->state_ = TIMER_STARTED;
00363:     global_interrupt_enable (interrupt_level);
00364: }
```

图 24.31

第 309 行定义了几个需要使用的局部变量。第 312 行关闭中断为后面遍历链表做准备。第 318 行对“桶”中的 level_变量加一并记录在 level 局部变量中。第 319 行对“桶”的 reentrance_变量加一，表示目前正对“桶”进行访问。count 变量被用于统计 timer_insert()函数在一次关闭中断的情形下连续遍历节点的个数，在第 337 行对其加一。程序运行到第 337 行说明已经完成

了一个节点的遍历且还没有为被启动的定时器找到插入点。第 339 行判断 count 变量的值是否小于 CONFIG_INTERRUPT_FLASH_FREQUENCY，如果小于则说明还可以再遍历下一个节点，因此在第 339 行直接调用 continue 语句。

程序运行到第 342 行，说明 count 的值大于等于 CONFIG_INTERRUPT_FLASH_FREQUENCY。第 342 行先对它进行清 0，因为后面马上要恢复中断了。第 343 行恢复一次中断，这会给已发生的中断一次被处理的机会，也正因为这一操作可以提高中断的响应实时性。

只有所有中断被处理完了后，第 346 行的代码才有机会执行。在中断被再一次关闭后，必须检查中断恢复（第 343 行）和关闭（第 346 行）期间所遍历的链表是否发生了更改，判断的机理就是根据局部变量 level 的值。如果这个值与“桶”中记录的值相同，则说明期间链表没有被更改，在这种情形下可以继续对链表进行遍历，即从第 352 行开始运行。反之，说明链表被更改过了，此时需要重新遍历链表。在重新查找插入点之前，在第 348 行先记录下“桶”中 level_变量的新值，并在第 349 行更新统计信息。

程序运行到了第 353 行说明成功地将定时器放入了“桶”中。在第 359 行需要对“桶”的 reentrance_变量进行减一操作，并通过判断它的值在减一后是否为 0 来决定是否对“桶”中的 level_变量进行清 0 操作。level_变量只有在“桶”当前没有任务或中断服务程序对它进行变更的情形下才会被置 0，而 reentrance_变量的值正是代表当前有多少任务或中断服务程序在更改该“桶”。

除了 timer_insert()函数需要更改外，其他几个会更改链表的函数也需要做相应的修改。图 24.32 是更改后的 timer_free()函数的实现。

```
00269: error_t timer_free (timer_handle_t _handle)
00270: {
00271:     interrupt_level_t level;
00272:
00273:     level = global_interrupt_disable ();
00274:     if (is_invalid_handle (_handle)) {
00275:         global_interrupt_enable (level);
00276:         return ERROR_T (ERROR_TIMER_FREE_INVHANDLE);
00277:     }
00278:     if (TIMER_STARTED == _handle->state_) {
00279:         timer_handle_t next;
00280:
00281:         if (g_timer_next == _handle) {
00282:             g_timer_next = (timer_handle_t) dll_next (
00283:                 &g_bucket_firing->dll_, &_handle->node_);
00284:         }
00285:         next = (timer_handle_t)dll_next
00286:         (&g_buckets [_handle->bucket_index_].dll_, &_handle->node_);
00287:         if (0 != next) {
00288:             next->round_ += _handle->round_;
00289:         }
00290:         dll_remove (&g_buckets [_handle->bucket_index_].dll_, &_handle->node_);
00291:         if (g_buckets [_handle->bucket_index_].reentrance_ > 0) {
```

```
00292:            g_bucket_firing->level_ ++;
00293:        }
00294:    }
00295:    else {
00296:        dll_remove (&g_inactive_timer, &_handle->node_);
00297:    }
00298:    _handle->magic_number_ = 0;
00299:    dll_push_tail (&g_free_timer, &_handle->node_);
00300:    global_interrupt_enable (level);
00301:
00302:    return 0;
00303: }
```

图 24.32

图中的第 291～293 行是新增的代码，即在释放一个定时器时，如果发现定时器所在的“桶”正在被更改，则需要对“桶”中的 level_变量加一。对 level_变量的更改最终会让 timer_insert()函数感知，使其重新遍历链表。

timer_free()函数中新增的代码在 timer_fire()和 timer_stop()函数中也能找到，在此不再一一列出，读者可以通过文件比较器发现这些变更。由于为“桶”增加了 redo_统计变量，因此在 timer_dump()函数的实现中，需要将这个统计变量进行显示，而代码也做了一点小小的改动，读者同样可以自行检查源代码辨别出这一改动。

这里改善实时性的方法就是将一次性长时间关闭中断的行为变成多次短时间。改善实时性并非一件易事，它需要我们对中断、任务、调度器有很清晰的认识，并在头脑中能以“并行”的方式思考多任务和中断问题，这对我们的能力是一个较大的挑战。

尽管这里只介绍了如何改进中断响应的实时性，但这一方法也可以运用于改善任务实时性。

24.7 任务回调定时器

任务回调定时器的实现可以在中断回调定时器的基础上，通过使用 21.4 节所介绍的消息队列来实现。

24.7.1 程序实现

任务回调定时器需要有一个专用于调用定时器回调函数的任务，因此在定时器管理模块中需要在系统初始化时创建任务。module_timer()函数的改动如图 24.33 所示。

```
00044: #define CONFIG_TIMER_TASK_STACK_SIZE     2048
00045: #define CONFIG_TIMER_TASK_PRIORITY       8
00046: #define CONFIG_TIMER_QUEUE_SIZE          1024
00047:
00056: typedef struct {
00057:     statistic_t notimer_;
00058:     statistic_t traversed_;
```

```
00059:     statistic_t abnormal_;
00060:     statistic_t queue_full_;
00061: } timer_statistic_t;
00062:
00077: static queue_handle_t g_timer_queue;
00078:
00165: error_t module_timer (system_state_t _state)
00166: {
00167:     usize_t idx;
00168:     error_t ecode;
00169:     static task_handle_t handle;
00170:
00171:     if (STATE_UP == _state) {
00172:         STACK_DECLARE (stack, CONFIG_TIMER_TASK_STACK_SIZE);
00173:         QUEUE_BUFFER_DECLARE (buffer, sizeof (timer_handle_t),
00174:             CONFIG_TIMER_QUEUE_SIZE);
00175:
00176:         ecode = queue_create ("Timer", &g_timer_queue, buffer,
00177:             sizeof (timer_handle_t), CONFIG_TIMER_QUEUE_SIZE);
00178:         if (0 != ecode) {
00179:             return ecode;
00180:         }
00181:         ecode = task_create (&handle, "Timer",
00182:             CONFIG_TIMER_TASK_PRIORITY, stack, sizeof (stack));
00183:         if (0 != ecode) {
00184:             return ecode;
00185:         }
00186:         ecode = task_start (handle, task_timer, g_timer_queue);
00187:         if (0 != ecode) {
00188:             return ecode;
00189:         }
00190:     }
00191:     else if (STATE_DOWN == _state) {
00192:         if (0 != task_delete (handle)) {
00193:             console_print ("Error: cannot delete task \"Timer\"");
00194:         }
00195:         if (0 != queue_delete (g_timer_queue)) {
00196:             console_print ("Error: cannot delete queue \"Timer\"");
00197:         }
00198:     }
00199:     else if (STATE_DESTROYING == _state) {
00200:         for (idx = 0; idx <= BUCKET_LAST_INDEX; ++ idx) {
00201:             (void) dll_traverse (&g_buckets [idx].dll_, timer_check_for_each, 0);
00202:         }
00203:         (void) dll_traverse (&g_inactive_timer, timer_check_for_each, 0);
00204:     }
00205:     return 0;
00206: }
```

图 24.33

第 44 行定义了定时器回调任务的栈空间大小，在这里定义的是 2048×4 字节，其大小可根据实际应用进行调整。第 45 行定义了回调任务的优先级。第 46 行定义了消息队列存放消息的最大数目。

第 60 行为模块的统计信息增加了 queue_full_统计变量，这个变量记录的是因为消息队列满而丢失消息的个数。第 77 行定义的是消息队列的句柄。

第 171～190 行对应的是系统进入“启动”状态的情形，在这种情形下需要创建定时器回调任务“Timer”和消息队列。注意第 173 行对消息队列缓冲区的定义，每一个消息其实就是一个定时器的指针。第 191～198 行对应于系统进入“关闭”状态的情形，在这种情形下需要将任务和消息队列删除。

回调任务的函数实现如图 24.34 所示。第 157 行任务通过接收消息的形式等待到期的定时器。一旦 queue_message_receive()函数返回，就说明出错或者成功收到定时器到期的消息。第 161 行调用定时器的回调函数。

```
00151: static void task_timer (const char _name [], void *_p_arg)
00152: {
00153:     queue_handle_t queue = (queue_handle_t)_p_arg;
00154:     timer_handle_t handle;
00155:
00156:     for (;;) {
00157:         if (0 != queue_message_receive (queue, 0, &handle)) {
00158:             console_print ("Error: task \"%s\" cannot recieve message", _name);
00159:             (void) task_suspend (task_self ());
00160:         }
00161:         handle->callback_(handle, handle->arg_);
00162:     }
00163: }
```

图 24.34

另一处大的改动位于 timer_fire()函数内，如图 24.35 所示。

```
00091: void timer_fire ()
00092: {
00093:     ......
00113:         else {
00114:             // hooray, the timer is expired
00115:             dll_remove (&g_bucket_firing->dll_, &handle->node_);
00116:             dll_push_tail (&g_inactive_timer, &handle->node_);
00117:             handle->state_ = TIMER_STOPPED;
00118:             global_interrupt_enable (level);
00119:             if (TIMER_TYPE_INTERRUPT == handle->type_) {
00120:                 handle->callback_(handle, handle->arg_);
00121:             }
00122:             else if (TIMER_TYPE_TASK == handle->type_) {
00123:                 timer_message_send (handle);
00124:             }
00125:             level = global_interrupt_disable ();
00126:         }
00127:         ......
00138: }
```

图 24.35

在处理到期的定时器时，第 119～124 行先判断定时器的类型，以决定是采用中断回调还是任务回调。对于任务回调定时器，timer_message_send()函数会被调用，以便将到期定时器的

指针当做消息发送给“Timer”任务。timer_message_send()函数的实现如图 24.36 所示。

```
00080: static void timer_message_send (const timer_handle_t _handle)
00081: {
00082:     if (0 != queue_message_send (g_timer_queue, &_handle)) {
00083:         interrupt_level_t level;
00084:
00085:         level = global_interrupt_disable ();
00086:         g_statistics.queue_full_ ++;
00087:         global_interrupt_enable (level);
00088:     }
00089: }
```

图 24.36

timer_message_send()函数调用 queue_message_send()发送消息时如果出现错误，那有可能是消息队列已满，为此在第 86 行更新统计信息。

24.7.2　timerv3 示例程序

timerv3 示例程序的源代码与 timerv1 几乎相同，唯一的区别是创建定时器时使用的是 TIMER_TYPE_TASK 参数，而不是 TIMER_TYPE_INTERRUPT。其运行结果从图 24.37 中可以找到。

```
/embedded/build
$ ./release/timerv3.exe

System is going to be up

Info: press Ctrl+C to terminate!
Info: timer "One Second" is expired [in "Timer" task]
Info: timer "One Second" is expired [in "Timer" task]
Info: timer "One Second" is expired [in "Timer" task]
Info: timer "One Second" is expired [in "Timer" task]
Info: timer "One Second" is expired [in "Timer" task]
Info: timer "One Second" is expired [in "Timer" task]
Info: timer "One Second" is expired [in "Timer" task]
Info: timer "One Second" is expired [in "Timer" task]
Info: timer "One Second" is expired [in "Timer" task]
Info: timer "One Second" is expired [in "Timer" task]

Summary
-------
  Supported: 128
  Allocated: 4
  .BSS Used: 7460

Statistics
----------
    No Timer: 0
    Abnormal: 0
   Traversed: 165
  Queue Full: 0

Bucket Details
```



图 24.37

24.8 小结

软件定时器存在中断回调与任务回调两种，它们各具不同的特点。前者由于回调函数是在中断状态被调用的，因此具有很好的响应性，但可能造成比“滴答”中断优先级低的中断出现中断延时；后者不会产生对中断的延时处理，但定时器的响应性差一点。

在使用定时器时，需要注意定时器回调函数的实现。不论是中断回调定时器还是任务回调定时器，过于耗时的回调函数实现有可能造成在一个正常的中断时间周期内无法完成所有到期定时器的处理。因此，应力求让中断回调函数的实现简单高效。对于需要长时间处理的内容，可以考虑通过消息队列等形式，转移到相应的（应用层）任务去处理。

本章也展示了如何通过设计去改善实时性，关注设计中的实时性问题对工程师的能力要求是一个很大的挑战。

练习与思考

假设“滴答”的周期是 10 毫秒，如果因为不良设计使得在 10 毫秒内无法完成所有到期定时器的回调处理，那会造成定时不精确问题[③]。在这种情形下，中断回调定时器与任务回调定时器的精度是如何受影响的？如何通过设计来预防这种问题呢？

③ 在极端情况下，如果某一定时器的回调函数需要花费的时间大于 10 毫秒时，这种设计就会导致问题。

第 25 章 ClearRTOS“实时”操作系统

前面几个章节开门见山地介绍了 ClearRTOS 中的任务管理、任务同步与通信、内存管理、设备管理和定时器管理，而没有介绍 ClearRTOS 的设计原则、源程序目录规划等。在本章，我们就这些内容进行补充说明。

25.1 设计原则

设计 ClearRTOS 的第一个原则是简单。正如 ClearRTOS 中名字中所包含的“Clear”那样，作者尽量将 ClearRTOS 设计得清晰易懂，致力于追求简单而不失优雅。比如，各模块避免使用动态分配内存的方式，而是采用定义全局数组的方式预留所需的内存资源，这就是从设计简单性考虑而做出的选择。其他还有许多地方，相信读者在阅读 ClearRTOS 时能感受到。

第二个原则是让各模块所管理资源的数量具有可配置性。比如，对于任务管理模块，可以通过配置 task_bitmap_t 数据结构的方式尽量减少它所占用的内存空间。如果某应用程序只需 5 个任务，那么就可以通过配置 task_bitmap_t 数据结构使 ClearRTOS 最多支持 8 个任务。在前面也提到了，由于信号量和互斥锁都使用了 task_bitmap_t 数据结构，对 task_bitmap_t 的精确配置将进一步节约信号量和互斥锁所占用的内存空间。由于每一个资源都是可配置的，这就给各种应用程序充分定制的机会，使得内存资源的使用更高效，这对于嵌入式操作系统来说是很重要的内容。

第三个原则是让其具有高可查错性。高可查错性意味着需要统一的错误管理方法，也需要尽可能地收集各种统计信息，以便更好地了解各种资源的使用状态。这些内容读者在 ClearRTOS 的实现中都能找到。还有，每一个模块都设计成在系统终止化时检查是否存在资源泄漏问题，也是从可查错性方面考虑的。就作者的观点来看，嵌入式操作系统的设计不应当只局限于操作系统的范畴，更应站在应用程序的角度提供一些统一的方法，这一点与传统操作系统的设计有很大的区别。这有点将平台与框架的设计思想融入到了操作系统的设计之中。

25.2 源程序目录管理

位于光盘中 Project/embedded 目录下的程序由于需要反映各模块的设计变迁，因此不少模

块存在多个版本，各版本通过 v1、v2 等子目录名进行区分。到了本章，ClearRTOS 的实现都已完成了，因此完全可以对 ClearRTOS 的目录进行重组，以使它更具系统性。光盘中 Project/ClearRTOS 目录中的程序正是重组后的结果，其目录结构如图 25.1 所示。

从图中读者可以看出，除了 code/platform/kernel 目录及其子目录外，其他的目录结构与 embedded 项目几乎一样。后面我们将集中探讨 kernel 目录及其子目录的组织。

一个操作系统的源代码管理需要体现哪些概念呢？第一个想到的概念是处理器。现在的处理器存在多种架构，有来自 Intel 的 x86 和 x86-64 架构，也有来自 ARM 公司的 ARM 架构，等等。由此看来，需要将处理器按架构和型号进行管理，这正是 kernel 目录下存在 arch 目录的缘故。各类架构在 arch 目录下将占据一个子目录，比如 x86 就是 arch 目录的一个子目录。

只要是属于同一架构的处理器，它们之间都存在一定的共性。例如，只要是 x86 处理器，不论什么型号，它们对于 I/O 端口的操作方法和指令都是相同的。为此，在每一个架构的目录下面有一个 share 子目录，用于存放同一架构处理器所共享的代码。share 目录下的代码所编译出来的库名为 libarch.a。另外，每一个处理器型号也在所属架构的目录中占有一个子目录，比如在图 25.1 的 x86 目录下就有一个名为 simulator 的子目录，用于表示运行于 Linux 操作系统之上的一个虚拟处理器。与处理器相关的程序，都应放入处理器目录中。各型号处理器目录下的代码所编译出来的库名为 libcpu.a。由于对于每一个嵌入式系统其处理器都是确定的，尽管图 25.1 中各处理器目录所编译出来的库名都将是 libcpu.a，但是一个产品只可能使用其中的一个目录，因此不存在库冲突问题。

操作系统不论是运行于哪一个处理器之上，一定存在一部分代码与处理器是无关的。比如内存管理、定时器管理等都可以设计成与处理器无关。任务管理模块可能很具典型性，它的情景管理部分的代码与处理器是紧密相关的，但任务调度器及其他任务控制函数却与处理器是无关的。因此，context.c、save.S 和 restore.S 都被放到了处理器的目录中（即 x86/simulator 目录）。对于操作系统中与处理器无关的代码，将被放入 kernel/core 目录中，这一目录中的程序所编译出来的库名为 libcore.a。

操作系统的源代码管理需要体现的第二个概念是驱动程序。驱动程序的存放位置分两种情形。如果驱动程序是与处理器相关的，则应放入 arch 目录之下处理器所对应的子目录中；否则，应当在 kernel/driver 目录之下为之建立一个独立的子目录。比如，如果某一嵌入式产品中存在一块来自 Realtek 的 RTL8306 以太网交换芯片，那么它的驱动程序应当放入 kernel/driver/switch/rtl8306 目录中。这是因为 RTL8306 这块芯片可以被运用到基于任何处理器的嵌入式产品中，因此它应当被独立出来进行管理。

由于驱动程序大多需要进行 I/O 操作，为了保证放入 kernel/driver 目录下的驱动程序能与 ClearRTOS 所支持的任一处理器一同工作，必须使用同样的函数进行 I/O 操作，也就是说，所有的处理器应定义相同的 I/O 操作函数。I/O 操作函数的实现代码应当放入 arch 目录之下处理

器架构所对应的 share 子目录中。

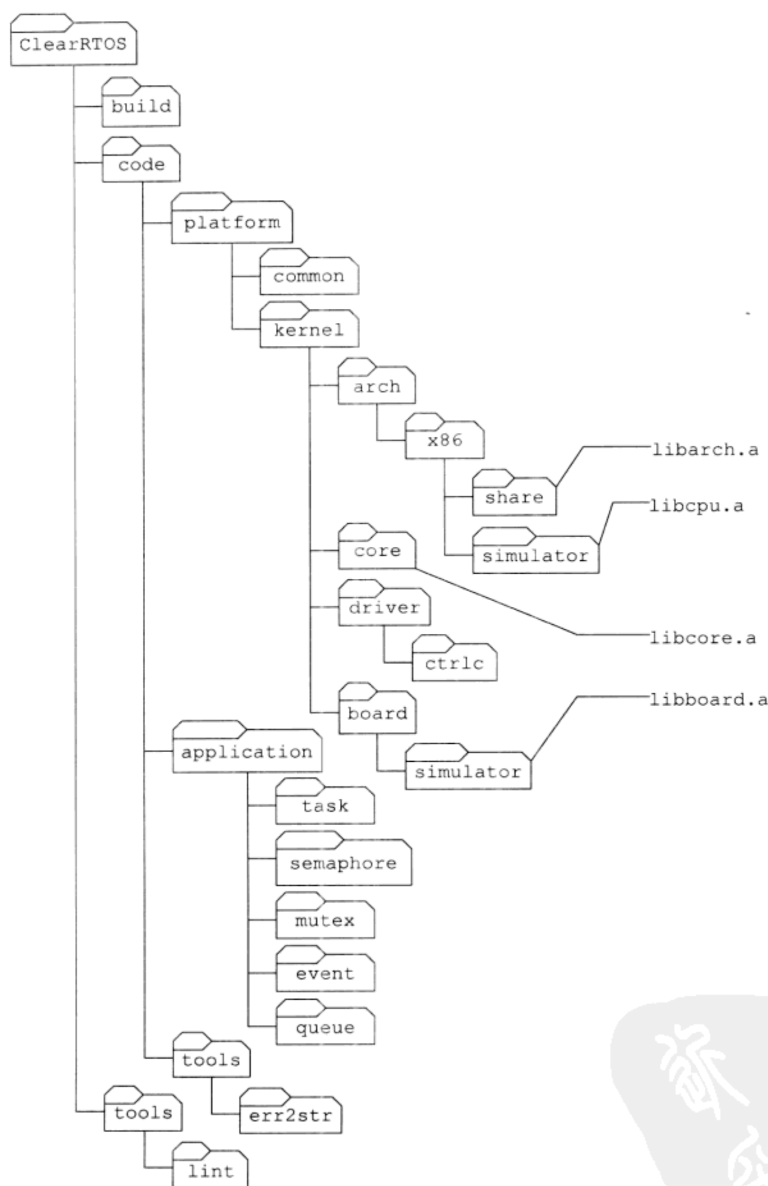


图 25.1

操作系统的源代码管理需要体现的第三个概念是处理器板。任何一个嵌入式系统都存在处理器板的概念，整个嵌入式系统有可能是由一块板组成的，也有可能是由多块板组成的。无论如何，各板卡都是由处理器和各种外围芯片组成的。因此，处理器板决定了操作系统应当包含

哪个处理器的库和哪些驱动程序。kernel/board 目录正是用于存放处理器板所特有的代码的。

当前，kernel/board/simulator 用于表示运行于 Linux 操作系统之上的虚拟板卡。在现实的嵌入式系统中，每一块板卡都应在 kernel/board 目录下占有一个子目录。在目前的 ClearRTOS 中，kernel/board/simulator 目录只存放了 inventory.c 文件，这个文件中实现了 device_registration_main()函数。kernel/board 目录下各子目录中的代码所编译出来的库名将是 libboard.a。

由于图 25.1 中并没有列出各目录下的具体文件，读者可以通过浏览各目录中的文件内容以更好地理解 ClearRTOS 的目录管理。

当需要增加与操作系统相关的程序文件时，通过图 25.2 能帮助读者将文件放入正确的目录中。请注意，该图是以运行于 Linux 操作系统之上的 ClearRTOS 为例的。

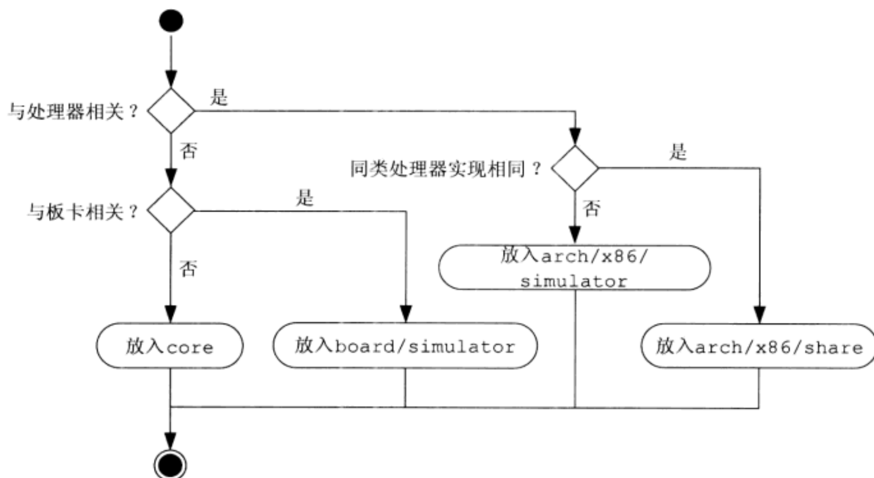


图 25.2

25.3 让 Makefile 体现概念

为了进一步方便 Makefile 的维护，在 Makefile 中也引入了架构、处理器类型和板卡这三个概念。图 25.3 示例说明了 build 目录下更改后的 Makefile，其中的粗体部分标记出了 embedded 项目与 ClearRTOS 项目之间 Makefile 的区别。

```
00001: ARCH = x86
00002: CPU = simulator
00003: BOARD = simulator
00004:
00005: ROOT = $(realpath ..)
00006:
00007: COMMON_DIRS = \
```

```
00008:     $(ROOT)/code/tools/err2str \
00009:     $(ROOT)/code/platform/common/src \
00010:     $(ROOT)/code/platform/kernel/arch/$(ARCH)/share/src \
00011:     $(ROOT)/code/platform/kernel/arch/$(ARCH)/$(CPU)/src \
00012:     $(ROOT)/code/platform/kernel/core/src \
00013:     $(ROOT)/code/platform/kernel/driver/ctrlc/src \
00014:     $(ROOT)/code/platform/kernel/board/$(BOARD)/src \
00015: ......
00055:
00056: .PHONY: release debug clean unitest test force creport scheck dcheck dreport touch
00057: release debug unitest clean scheck:
00058:     @set -e; \
00059:     for DIR in $(MAKE_DIRS); \
00060:         do \
00061:             cd $$DIR && $(MAKE) -r ROOT=$(ROOT)
                       ARCH=$(ARCH) CPU=$(CPU) BOARD=$(BOARD) $@; \
00062:         done
00063:     @set -e; \
00064: ......
```

图 25.3

第 1～3 行分别定义了对应于三个概念的变量，并将其设置为相应的值。图中所设置的架构类型为 x86，处理器型号为 simulator，以及板卡的型号也为 simulator。第 10、11 和 14 行分别通过引用这三个变量来指定相应的目录，而没有采用写死的方式。在第 61 行调用其他目录的 Makefile 时将 ARCH、CPU 和 BOARD 三个变量的值一并传入。

图 25.4 示例说明了 task.exe 所对应的新的 Makefile，其中的改动位于第 6 行，即通过引用 ARCH 和 CPU 两个变量来指定具体的头文件目录。

```
00001: EXE = task.exe
00002: LIB =
00003:
00004: INCLUDE_DIRS = \
00005:     $(ROOT)/code/platform/common/inc \
00006:     $(ROOT)/code/platform/kernel/arch/$(ARCH)/$(CPU)/inc \
00007:     $(ROOT)/code/platform/kernel/core/inc \
00008:
00009: LINK_LIBS = core board ctrlc cpu arch common
00010:
00011: include $(BUILD)/c.rule
```

图 25.4

25.4　实现集中配置

为了更加方便地配置 ClearRTOS，我们很有必要为之创建一个集中的配置文件。该配置文件并不需要包含 arch 和 board 两个目录之下的内容，而只需集中于 core 目录。新增的 config.h 文件如图 25.5 所示。

ClearRTOS/code/platform/kernel/core/inc/config.h

```
00026: #ifndef __CONFIG_H
00027: #define __CONFIG_H
00028:
00029: // for Task Bitmap
00030: // Summary:
00031: // Supported bits is calculated by "CONFIG_MAX_BITMAP_ROW * CONFIG_MAX_BIT_PER_ROW".
00032: // The max bit supported by this module is 1024 with below configuration:
00033: //     typedef unsigned int task_bitmap_row_t;
00034: //     CONFIG_MAX_BITMAP_ROW = 32;
00035: //     CONFIG_MAX_BIT_PER_ROW = 32;
00036: // The min bits supported by this module is 8 with below configuration:
00037: //     typedef unsigned char task_bitmap_row_t;
00038: //     CONFIG_MAX_BITMAP_ROW = 1;
00039: //     CONFIG_MAX_BIT_PER_ROW = 8;
00040: #define CONFIG_MAX_BITMAP_ROW                  4
00041: #define CONFIG_MAX_BIT_PER_ROW                 32
00042: typedef unsigned int task_bitmap_row_t;
00043:
00044: // for Task Stack
00045: typedef unsigned int stack_unit_t;
00046:
00047: // for Idle Task
00048: #define CONFIG_IDLE_TASK_STACK_SIZE            1024
00049:
00050: // for Task Variable
00051: #define CONFIG_MAX_TASK_VARIABLE               32
00052:
00053: // for Task Hook
00054: #define CONFIG_MAX_TASK_CREATE_HOOK            8
00055: #define CONFIG_MAX_TASK_SWITCH_HOOK            8
00056: #define CONFIG_MAX_TASK_DELETE_HOOK            8
00057:
00058: // for Sync Module
00059: #define CONFIG_MAX_QUEUE                       8
00060: #define CONFIG_MAX_MUTEX                       32
00061: #define CONFIG_MAX_SEMAPHORE                   32
00062:
00063: // for Memory Pool
00064: #define CONFIG_MAX_MPOOL                       2
00065:
00066: // for Driver Management
00067: #define CONFIG_MAX_DRIVER                      8
00068:
00069: // for Timer Management
00070: #define CONFIG_TICK_DURATION_IN_MSEC           10
00071: #define CONFIG_MAX_BUCKET                      13
00072: #define CONFIG_MAX_TIMER                       128
00073: #define CONFIG_TIMER_TASK_STACK_SIZE           2048
00074: #define CONFIG_TIMER_TASK_PRIORITY             8
00075: #define CONFIG_TIMER_QUEUE_SIZE                1024
00076: #define CONFIG_INTERRUPT_FLASH_FREQUENCY       2
00077:
00078: #endif
```

图 25.5

25.5 改进与移植

目前的 ClearRTOS 只支持基于优先级的调度，这种方式对于大多的嵌入式系统是够用的。

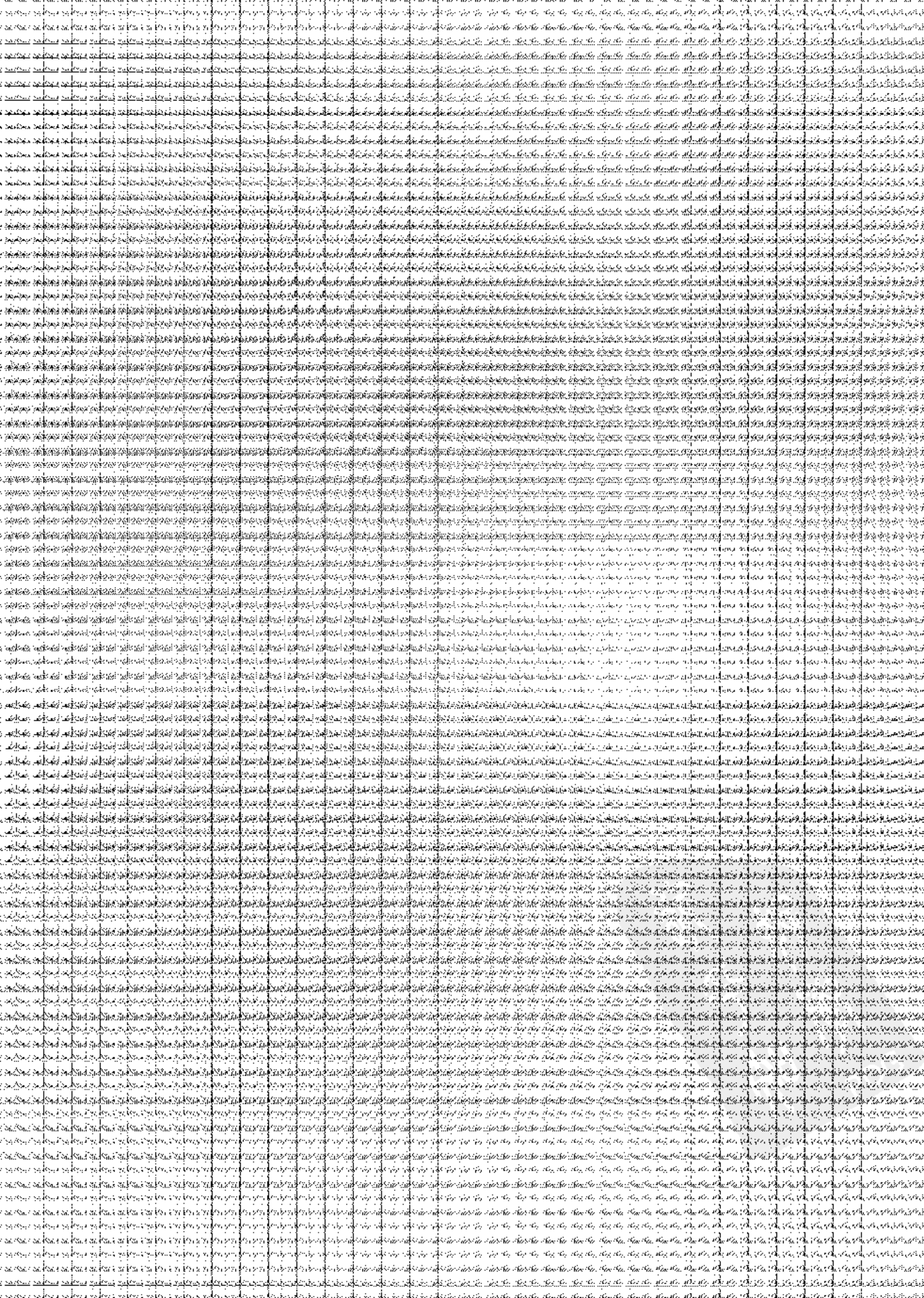
但是，一个生命力强大的实时操作系统，应提供更多类型的调度算法以适应不同应用的需要。比如，将 ClearRTOS 改造成同一优先级的任务可以有多个，并引入时间片的调度算法；让队列支持先进先出的任务等待方式；等等。作者认为，ClearRTOS 目前所提供的环境，足以让读者可以根据自己的想法去尝试设计自己的调度算法。因为 ClearRTOS 所创造的环境已使得开发实时操作系统如同开发一般的应用程序一样方便。

除了引入更多的调度算法外，还可以考虑将 newlib 库改造成能与 ClearRTOS 协同工作。对于一个适用的操作系统来说，丰富的 C 函数库资源也是很重要的一项指标。

前面的章节提到，本书中的 ClearRTOS 只提供了字符设备的驱动模型。可以考虑引入块设备的驱动模型使得 ClearRTOS 支持硬盘、闪存等设备，进而引入文件系统；也可以考虑引入帧设备的驱动模型以支持各种协议栈；等等。

最后，作者也希望读者尝试将 ClearRTOS 移植到不同的真实处理器、虚拟机或处理器模拟器上，将 ClearRTOS 变成一个真正的实时操作系统，也希望借助 ClearRTOS 使之成为一个交流和学习的平台。作者在 Google Code 上创建了 ClearRTOS 开源项目，其网址是 http://code.google.com/p/clearrtos/。





质量保证篇

与软件开发过程中产品从无到有的创造所带来的兴奋与有趣相比，软件质量保证更多地让人觉得沉闷与枯燥，正因如此，质量保证方面的工作被不少工程师所忽视。也正因为忽视质量保证，使得软件行业的加班加点让人习以为常，认为这个行业是“青春饭”，这也使得高质高效编程变得遥不可及。

对于软件质量的理解需要我们先了解软件行业的特点，进而理解软件质量的保证需要系统性的方法论。而方法论离不开流程的遵守和工具的运用，第 26 章就这方面进行了阐述。另外，还指出构建质量保证方法论应采用“关键要素有形化”、“流程与工具的无缝整合”和“以单元测试为中心”这三种手段。

高质量软件的获得不能一味地依赖于设计和测试，还应关注编码，而这离不开软件工程师的编程好习惯。好习惯对于我们生活和工作的积极作用毋需多言，但在编程方面的现实状况却是那么不尽人意，因此还是很有必要单独用一个章节来事无巨细地谈一谈编程好习惯，这也正是增加第 27 章的缘由。

保证软件质量很有效的一个手段是单元测试，在软件行业也有测试驱动开发(TDD)这种思潮。但是，一听到单元测试会让不少工程师紧张，因为他可能怀有“单元测试无用”和“是体力劳动”这类偏见。之所以产生偏见，是因为没有真正体会到单元测试的好处，这是实施单元测试的方法不当而造成的。在第 28 章将深入地探讨单元测试的作用，以及如何有效地实施它，并且阐述了什么是可测试性设计。

质量保证离不开流程和工具所构建的方法论，从第 29 到 32 章将分别介绍代码覆盖、静态分析、动态分析和性能分析这些方法的运用，并向读者展示如何将这些方法所需的工具无缝地集成到编译系统中。这几章也可以看做是对 make 进行“活用”的延续。

第 33 章是对本篇的一个小小总结，将同读者一同回顾本篇所打造的软件开发环境——qBench，并寄希望于读者将其运用到自己的工作中。

第 26 章 质量保证导言

与软件开发过程中产品从无到有的创造所带来的兴奋与有趣相比，软件质量保证更多地让人觉得沉闷与枯燥。与质量保证相比，需求分析、设计和编码是相对容易做的事，因为所出现的差错能让我们觉得有劲可使加以弥补，这些错误表现为“明枪”。软件质量保证则很容易让人有劲使不出，乃至怎么也做不好，觉得是“暗箭难防”。之所以会出现这种状况，是因为没有深刻地理解软件质量保证是一个系统工程。高质量软件的获得并不意味着只要做好软件开发过程中的某一个或几个关键环节就行了，而是需要关注软件生命周期内的所有环节。

对于质量保证这一系统工程，它不能被简单地理解为“就是测试”。在进一步探讨这一话题之前，需要先了解软件开发的特点。

26.1 软件开发的特点

软件开发具有以下几个特点，这些特点最终导致软件质量的控制并不那么直观和容易。

26.1.1 脑力密集型工作

软件开发是脑力密集型工作，其中的不少活动因为只存在于软件工程师的大脑中，因而具有不可见性。因此，我们无法通过运用流程这样的方法将潜在的质量问题完全消除，这与工厂流水线生产条件下的质量保证完全不同。

大脑在处理事务时并不能完全保证其一致性，有时甚至会受情绪的影响。大脑的“善变”有它的好处，比如让我们更具创造性，也为我们的生活带来了更多的乐趣，但这对于软件质量保证却未必是一件好事。为了减小善变所带来的负面影响，培养良好的工作习惯是一条不错的途径。

26.1.2 实现不具唯一性

一个软件功能，尽管从使用者的角度来看都一样，但却可以有多种不同的实现方法，且不同的人、不同的开发团队所做出来的设计实现有可能完全不同。如果实现具有唯一性，那么质量就更容易被评估，也更容易找到改善点，但软件不属于这一列。

正因为软件的实现不具唯一性，这使得很难从林林总总的实现中找到哪一种更好，或者要找到那种“更好”所需付出的成本很高。

26.1.3 隐性成本高

与其他类型产品的开发相比，软件产品开发的隐性成本很高。所谓的隐性成本，是指在项目预算时并没有将其考虑在内，但它确实在将来的开发活动中会导致额外的成本开销。

一个软件项目的阶段性完成并不意味着它不会带来后续的成本，因为不同的实现（实现不具唯一性）所带来的软件稳定性和可维护性都不同。如果是不良实现，其所带来的隐性成本往往在项目初期预算时无法合理地被估计，这进一步又意味着什么呢？第一，这将导致难以对项目进行有效计划。这是显然的，因为看不到隐性成本的存在，在项目计划时就不会将其列入其中。第二则更为严重，由于隐性成本的不可见性很容易被忽视，进而不能掌握软件开发的特点，对软件开发过程中的困难也表现得不理解，甚至一味地认为只要投入时间、财力和人力就一定能开发出高质量的软件产品。

26.1.4 忽视的细节很容易被放大

软件开发过程中一个很小的细节很容易被放大。对于一个模块在设计时所留下来的小窟窿，哪怕认为微不足道，但它也很有可能演变成项目团队的沉重负担。例如，对于大型项目，如果大家随意地包含头文件，最后很有可能造成每一次项目编译都浪费不少时间去等待；修补一个缺陷时，由于觉得没有必要去除其中的一处冗余设计，却有可能最后落得难以维护；等等。

莫非定律在软件行业似乎总被佐证。有时候用“如履薄冰”来形容软件开发一点都不夸张，这也是为什么在 13.6.8 节提出在软件设计时要防止给他人留下犯错机会的原因之一。

26.1.5 质量难以评估

一个功能上完好的软件其设计未必就好，而设计不好则早晚会出问题，从而带来隐性成本。要真正地评估软件的质量需要从评估其设计质量着手，而这需要质量评估者很具专业性——对软件行业有深刻的理解和丰富的（编程）经验，以及对软件设计思想有充分认识。

设计质量评估所需的专业性正是因为实现不具唯一性这一特点决定的。

26.2 保证质量的关键要素

软件在没有发布之前的开发过程主要分为需求分析、设计、编码和验证四个阶段，最终的软件质量与这四个阶段的各自质量之间的关系用 C 语言来表达应是：

```
最终的软件质量 = 需求分析质量 && 设计质量 && 编码质量 && 验证质量
```

即，最终的质量来自于各阶段质量之“与”。只要其中一个环节质量差，产品的整体质量

就都将差。因此，每一个阶段的质量都起着决定性的作用，这一公式也体现了“软件开发无小事”这一道理。

以上四个阶段的质量将引出以下软件质量保证的关键要素。

26.2.1 完备的需求分析

需求分析的目的是让项目团队明白要做什么，是决定所开发出来的软件是“什么模样”。显然，完备的需求分析是高质量软件的前提。如果所开发出来的软件与用户所希望的并不一致，那不可能让用户说“这个软件的质量很好”。方向不对，软件设计得再“好”也无意义。需求分析失误所带来的开发成本是高昂的，这一点在《软件工程》这类书籍中都有提及。因此，整个行业对于需求分析的重要性具有足够的认识。当然，知道其重要性与如何获得完备的需求分析结果又是两回事，如何做好需求分析超出了本书的范畴。

需求分析出现失误有一个特点——它一定会暴露！只不过存在是暴露在软件开发过程中还是用户手中之别。因此，需求分析所造成的问题尽管严重，但它能被发现并能得到项目团队的重视而修复。当然，不同阶段发现这类问题所花费的成本有所不同。

26.2.2 高质量的设计

设计阶段是通过设计方法找出软件实现更好的方法，注意这里是“更好”两个字，而不是强调最好。设计是软件质量之本，其重要性决定了在第 13 章花了整整一章去探讨它，将其列在这里是为了章节的完整性。

26.2.3 编程好习惯

设计阶段输出的结果就是设计蓝图，但好的蓝图并不能保证最后的质量一定就好。拿建造房子打个比方，图纸设计得再科学和合理，如果建造时使用的材料不过关，那最终的房子一定好不了。软件开发中的“建筑材料”是工程师所编写的代码，“建筑材料”的质量需要通过良好的编程习惯去保证，在第 27 章就编程好习惯进行了探讨。

在现实的项目中，设计有可能与编码会有一定的糅合，即通过一定的编码工作来辅助设计。这种实践方式并不影响这里将设计与编码分为两个质量保证的关键要素。

26.2.4 充分的验证

验证很容易让人想到质量保证的常用方法——测试，但验证应当包含更多的内涵，比如求证软件需求是用户所希望的就是其中的一个。

对于验证的理解仍需要拿房屋的建造作比方以便加深理解。在房屋的建造过程中，当建筑材料到了工地以后需要对其进行检验，以保证它的质量是合格的。对应于软件开发，这个阶段

就是单元测试，当软件工程师编写了代码以后需要通过单元测试去验证。房子建造好了以后，还得对房子进行整体的验收以确保其最终是合格的。在软件开发过程中，软件集成测试就如同房子在建造好后的验收。

从上面的比方能得出这样的结论。第一，在软件开发过程中单元测试是必不可少的。它的缺少如同将没有检验过的建筑材料用于房屋建造一样。第二，单元测试工作应当在集成测试工作之前完成。有的项目在一开始时并没有包含单元测试流程，但后来发现需要增加这个环节，于是出现了集成测试完成了以后再进行单元测试这种情形。出现这种状况还是有点怪怪的，这如同房子已造好了，再将墙打掉去检查里面的砖是否是好的一样。这种行为的勇气是可佳的，但是如果尽早地在项目中部署单元测试，就能避免这种“劳民伤财”之事的发生。

集成测试（包括开发集成和系统集成）在软件行业被广泛采用以保证软件质量，但单元测试对于软件质量保证的重要性在整个行业还缺乏广泛而深刻的认识，这一工作更多地被当做是负担而不是一种有效的质量保证手段。在第 28 章将进一步探讨这一被忽视的、有效的质量保证方法。

26.2.5 必要的流程

对于流程可能每个项目团队的理解都有所不同，但无论如何，必要的流程对于软件质量的保证是一种保障。需要设计审查以确保设计质量；需要将单元测试、静态分析和动态分析作为流程以控制编码质量；需要代码版本控制流程来保证并行开发的实施；需要需求管理流程来确保开发部门开发出用户所需的产品，以及保证测试部门没有遗漏地对产品加以验证；等等。

一个软件项目是否存在必要的流程是检验项目团队是否是“手工作坊”的重要指标。无论项目的规模大小，必要的流程都是不可或缺的，否则只能完全靠人的主观意识去行事。流程的目的在于帮助避免没有必要的“低级错误”，或者说是通过外在的、具体的方法帮助项目团队维持良好的工作习惯。

当然，流程不是万能的。建立与自己的项目相匹配的流程才有意义，否则会出现团队为了流程而流程，进而得出“流程无用论”。如果发现流程的运用使得团队没有时间去做真正有意义的事情，那应当进行流程上的反省，以找到其中没有价值的内容并改进。流程的运用不在于只执行流程所规定的动作，而在于明白其背后的思想，以及站在开发团队的角度去理解每一个动作所带来的有利于按时交付高质量软件的益处。

敏捷软件开发现在很受欢迎，其核心思想包含了对流程的简化。注意，简化的目的不是为了偷懒，而是为了让我们更能应对开发过程中的不确定性以保证项目的成功。作者认为，运用敏捷思想指导软件开发需要较高的水平，它的本意是需要使用者先理解软件行业的特点，再进行简化。可是，有些项目团队其实根本没有什么流程，只是因为没有才将自己称为运用了敏捷思想指导软件开发，更有甚者将敏捷方法中的片面内容当做了不做某事的借口。

26.2.6 合适的工具

可以说，软件开发离不开工具的支撑，否则项目团队很有可能是另一种形式的"手工作坊"。对于设计，需要 UML 工具；对于代码的静态分析和动态分析，需要有相应的分析工具；对于代码版本控制，需要版本控制工具；对于需求管理，又离不开需求管理工具；等等。

通过一定的开发工具来帮助开发出高质量软件已是行业的共识。有趣的是，一个工具在一定的项目规模下能很好地发挥作用，可是当规模大到一定程度时工具的缺点就显得很明显，进而可能造成工具反而成了开发团队的另一个沉重包袱。

由此看来，合适的工具对于项目团队更具有意义。"合适"在这里应当理解为易用、够用。现在市面上有各种各样的工具，但不少工具厂商在宣传时都大力强调自己的优点，而没有告知其缺点和更具体的运用场合。此外，作者认为不少工具厂商有将工具复杂化之嫌，这些工具的概念很好但做得过于复杂。

工具不只在于会用，更在于理解它所解决的问题和对软件质量的意义。没有这方面的深刻认识，工具通常也不能很好地发挥效能。

26.2.7 言简意赅的文档

高质量软件包含高效地从事软件开发工作，高效工作离不开文档。文档类型包括设计文档、开发指南和测试手册等，它们都能帮助团队提高开发效率。

文档的重点在于阐述设计思想或工作方法，其着力点应当是"点到为止"地将问题说清道明。我们应以加强沟通和提高效率当做是写文档的目的，而非为了文档而写文档。

很多开发团队在敏捷开发思想的影响下，认为文档可有可无，作者认为这是对敏捷开发思想的一种误解。

除了设计文档很重要外，开发指南、测试手册这类文档的作用也不可小视。对于大型软件项目，其中存在很多的工作流程，包括如何准备开发环境、编译等。有了指导性文档以后，当项目有新人加入时，它们有助于节约培养新人的时间。指导性文档还有助于帮助项目团队沉淀开发活动中的知识。大型项目在开发过程中会出现很多非软件缺陷问题，采用文档记录这些问题有助于加速开发进程。

当然，写好的文档不应成为摆设，让文档真正有用的关键是倡导一种习惯于使用文档的团队文化。

26.3 质量保证需要系统性的方法论

软件质量保证是一个非常复杂的系统工程，前面谈到高质量软件的获得不能通过只做好软

件开发活动中的某一个或几个环节，当然更不可能在没有任何方法的情形下“意外地”获得，除非软件的规模非常小且所有开发工作是由一、二个能人完成的。

26.3.1　方法论 = 流程 + 工具

方法论是指为了保证软件质量，一个个整合在一起的极具可操作性的流程，且每一个小流程都有助于达到质量保证方面的某一目的。前面提到了，流程离不开工具的支撑，因为只有使用工具才能显著地提高流程的可操作性。

方法论有大小之别，应当视软件的规模和人员的多少进行量体裁衣。方法论不在于形式，而在于能达到软件质量保证领域中的特定目的。下面让我们看一看，一个软件质量保证方法论应当涵盖哪些方面。图 26.1 示例说明了软件项目中的关键活动，以及关键活动中需要用到的工具。为了简单化，图中并没有将流程之间的反复过程表达出来。

软件项目最原始的起点是用户的需要，需要经过了市场部门的识别后就变成了商业机会。一旦市场部决定抓住某一商业机会，相关部门将共同完成预算和制定开发计划。

接着系统工程（system engineering）部门根据用户的需要捕获需求。当需求被捕获了以后，如何对需求进行管理呢？为此，我们需要思考需求的管理是直接采用一个 Word 文档，还是采用专业的需求管理工具。需求管理不能简单地理解为在哪个地方存放需求描述这么简单，而是还得考虑开发团队和测试团队如何对需求进行跟踪。

为了使开发团队便于管理需求，我们需要将功能需求转换为软件需求。这需要对其分层，比如按层次的高低可以分为系统层、子系统层、设备层和模块层。显然，各层次间的需求是有关联的，低层次的需求来源于高层次，但将高层次的需求进行了细化。如何反映各层次之间的依赖关系是需求管理很重要的内容。在需求管理方面，来自 IBM 公司的 DOORS 被大型公司广泛采用。DOORS 是采用自然语言的方式描述需求的（但我们可以在 DOORS 中嵌入图片）。在 UML 被广泛普及的现在，采用 UML 管理需求也是大势所趋。

有了需求以后，开发部门开始设计工作。设计质量的重要性在第 13 章进行了说明。设计阶段最需要注意的是时间压力。作者的体会是，开发最耗时的部分就是设计阶段。设计阶段的时间如果不足，会因为没有思考清楚而容易造成概念不清，从而使得编码和测试阶段耗费更多的额外时间。其实，一旦设计做好了后编码和测试工作都相对省时。注意，不是投入的时间越多，设计质量就一定会越好，设计质量的保证除了需要时间更要有能胜任的人。

用于辅助设计工作的常用工具是 UML 工具，以及用于帮助思考所编写的原型代码。UML 虽然因为过于复杂而受到部分人士的批评，但它仍被很多行业广泛采用，它的衍生建模语言如 SysUML、BPMN 等也正蓬勃发展。功能强大的 UML 工具主要来自商业公司，强大是指工具能很好地遵循最新的 UML 规范。比如 Visual Paradigm 公司的 UML 工具就很不错，不论是可使用性还是对规范的遵循都很突出。除了 Visual Paradigm 的 UML 工具，来自 IBM 的 Rational

Software Modeler 也是不错的选择。在商业软件之外，开源的 StarUML 也被很多人采用，但是由于它不能紧跟规范的发展，所以它的表达能力存在局限性。

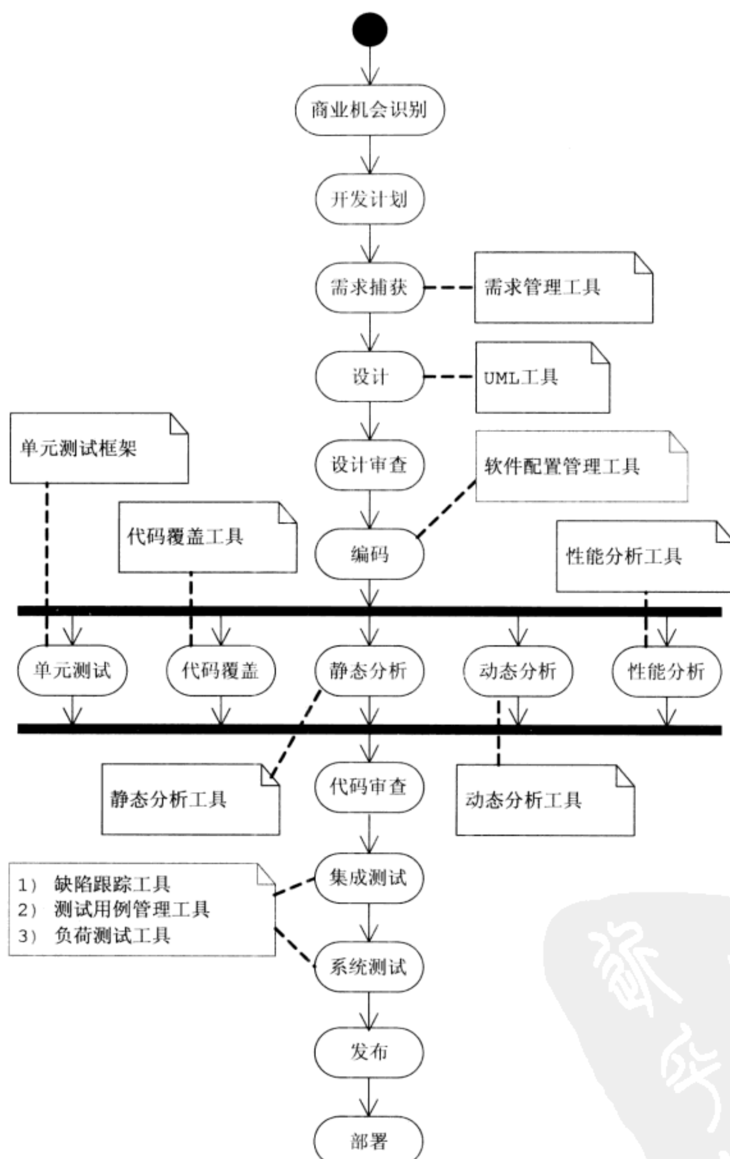


图 26.1

设计质量需要通过设计审查流程去保证。设计审查不同于代码审查，设计审查的重点在于检查设计概念，相对抽象。在设计审查时，如果设计者不能通过讲解设计概念让与会者了解设计意图，那很有可能表明设计质量并不高。

设计离不开编写设计文档，设计文档采用 Word 形式就足够了，文档中加入 UML 图进行设计表达则更好。另外，设计文档应当随着工作的进展而不断地被修订，每一次修订的目的应记录在修订历史表格中。

设计审查之后进入编码阶段。编码阶段在很多方面需要考虑方法问题。首先，需要考虑的是源代码如何管理，在软件行业这被称为软件配置管理（Software Configuration Management，SCM）。源代码版本控制工具是开发团队不可或缺的工具之一，没有有效的源代码管理工具，想要做到多人同时进行高效的编码活动是不可思议的。源代码版本管理工具可选择的范围较广，不论是商业软件还是开源软件。作者认为在大多数的软件项目中，使用开源的版本管理工具足以应付日常的开发工作，如 Subversion、Git 和 CVS。其次，版本管理除了选择工具之外，还需要制定好版本的命名规则，这也涉及到最终对外发行软件的版本问题。一个好的版本命名规则既能让开发工作更加有序，也能让外部人士感受到项目的开发工作是否专业。

编码的质量又如何保证呢？第一，工程师需要有良好的编程习惯，这能在第一时间保证所获得代码的质量。要让工程师养成良好的编程习惯，除了需要工程师自我培养外，开发团队也应努力创造这种氛围。

第二，进行单元测试（unit test）。单元测试是一种有效的质量保证方法，但是不少人对于这一点的认识并不深刻。第 28 章详细地介绍了什么是单元测试，以及如何将单元测试整合到开发环境中。

第三，通过代码覆盖保证单元测试效果。通过代码覆盖报告，我们可以知道一个模块的单元测试效果如何，是否存在重要的逻辑没有测试到的情形，进而提供决策信息以确定是否需要增加更多的单元测试用例。第 29 章讲解了如何将代码覆盖工具无缝地整合到开发环境中。

第四，引入代码静态分析（static analysis）和程序动态分析（dynamic analysis）。静态分析和动态分析分别在第 30 和 31 章进行专门介绍，在这两章中也将介绍如何将相应的工具无缝地整合到开发环境中。

第五，运用性能分析（profile）确保编写代码的执行效率。对程序执行效率的判断不应太主观，性能分析工具所出具的报告应成为代码优化的依据。在第 32 章就性能分析工具和将其无缝地整合到开发环境中做专门的介绍。

虽然编码质量的保证方法和工具在图 26.1 中被放在编码和代码审查之间，但这些工具的运用应当涵盖在编码阶段，或者说工程师在编码时应当交替使用这些工具去确保所编写代码的质量。

尽管在编码阶段有编程好习惯和各种工具的“保驾护航”，但是代码审查流程还是不能被跳过。每一个项目都有各自的特点，这将决定软件的架构、设计时应注意的具体事项等内容会有所不同，且这些不同不能被前面提到的质量保证工具所涵盖。因此，必须通过代码审查的方

式去发现潜在的问题。代码审查会议更多的是使用项目已沉淀的经验，看看代码是否遵守了成功经验或违背了最佳实践。值得一提的是，代码审查工作应当在使用了编码质量保证工具之后再进行，这样能显著地提高代码审查的效率，因为工具能发现的错误不应成为代码审查会议的焦点之一。

代码审查之后，需要将所开发的软件（模块）集成并进行集成测试。集成测试可以由开发工程师或测试工程师完成，但仍属于开发部门的工作内容[①]。集成测试并不要求完全使用真实的设备，而是可以使用模拟工具进行模拟测试（这是为了减小设备采购成本），这与系统测试有很大的不同。

集成测试需要使用缺陷跟踪工具提交测试时发现的缺陷，保证相关部门参与解决。缺陷可以是提交给系统工程部门的需求缺陷，也可以是提交给开发部门的程序实现缺陷。集成测试离不开编写测试用例，因此需要使用测试用例管理工具来管理它们。另外，集成测试还需要使用负荷测试工具对产品进行负荷测试，以阶段性地保证产品的性能。

集成测试一旦通过，产品就可以从开发部门移交给测试部门进行系统测试了。系统测试所搭建的测试环境与产品在用户手中的环境是非常相似的。完成系统测试后的产品就可以发布了。软件发布与代码版本管理是紧密相关的，因为代码版本与软件版本之间存在对应关系。发布工作除了准备好软件安装包外，还得准备好用户手册等文档。

发布后的软件可以被部署到用户现场。当用户发现问题时会联系技术支持部门寻求帮助，如果是软件缺陷，技术支持部门会通过缺陷跟踪工具提交缺陷记录。

最后，作者想指出质量保证方法论中一个容易被忽视的关键因素——时机。一个应当在早期部署的流程或工具如果被推迟到后期，则会像需求分析错误被遗留到软件开发后期那样产生巨大的、额外的成本开销。理论上，所有的流程和工具都应当在开发工作启动时就部署好，不适时机所引入的流程和工具很有可能最终让团队误认为“只是负担”和“没有效果”。

26.3.2 构建有效方法论的核心手段

有了工具和流程并不表示质量保证方法论就一定有效，还需要注意它们的可操作性和易用性。有效的质量保证方法论源于构建手段。

26.3.2.1 关键要素有形化

在保证软件质量的关键要素中，大体上可以将其分为有形的和无形的两大类。是否有形是指能否通过一定的具体方法施加影响以保证软件质量。比如，设计能力和编程好习惯一开始就不是一种有形的要素，而工具、文档却是有形的。

① 开发部门也可以设置测试团队。

将无形要素有形化是打造质量保证方法论的核心手段之一。对于好的设计思想这一无形的要素，可以通过抽取设计原则，以文档化的方式使其有形；对于编程好习惯也可以通过文档将其有形化；等等。有形化的要素容易在项目团队中被学习和实践，使这些要素在人的行为中体现“形”。

一个好的质量保证体系应尽可能地将无形的要素转换成有形的，以便减少其中难以控制的“艺术”成份而获得良好的可操作性。

26.3.2.2　流程和工具的无缝整合

对于质量保证方法论中的工具和流程，应尽可能地将它们与项目开发环境进行无缝整合，无缝整合的目的在于保证易用性。工具和流程只有易于使用，才能在项目团队中最大限度地发挥其价值。另外，将一些重复性的工作自动化，也是提高易用性的一种方式。

软件开发是一种将无形的需求有形化的过程，有形化后的产物从项目团队的角度来说就是代码。由于代码是无形需求的外在表现，因此代码的质量对于整个软件产品的质量具有决定性的意义。先不说代码在设计上做得如何，但无论怎样的设计所获得的代码都不应当包含编码错误。因此，为了保证代码的质量，需要通过运用工具和方法来找出其中存在的错误。图 26.2 示例说明了应与开发环境无缝整合的工具和方法。

无缝整合的结果是工程师能轻松地使用它们，并体会到其所带来的益处。不少工具和方法之所以不能被持久地运用而发挥作用大多是因为使用起来太麻烦，乃至工程师还没有尝到“甜头”就放弃了。软件产品的质量源于对每一个软件模块的质量把控，因此开发环境中无缝整合的工具和方法应被应用于每一个软件模块的编码过程，也只有这样质量管理才能落到实处。

26.3.2.3　以单元测试为中心

为了获得更好的易用性和效果，在构建质量保证方法论时应做到以单元测试为中心。请不要将“单元测试为中心”理解为“只要做好单元测试就能保证软件质量”。图 26.3 示例说明了哪些方法和工具应以单元测试为中心。



图 26.2



图 26.3

前面提到，软件产品质量的保证在很大程度上需要通过代码质量保证加以落实，而单元测

试能做到最小粒度的代码功能验证。单元测试需要通过设计测试用例保证代码尽可能多地被执行，这也正是代码覆盖、动态分析和性能分析所需要的。将代码覆盖、动态分析和性能分析以单元测试为中心进行整合，能实现通过进行单元测试“顺便”完成以之为中心的其他质量保证工作。

以单元测试为中心这一手段有点抽象，在第 29、31 和 32 章读者将看到是如何通过以单元测试为中心方便地完成代码覆盖、动态分析和性能分析的。

26.4 走出质量困境的指导性思想

首先，不论是管理者还是工程师对于软件设计的重要性都应有充分的认识。这是走出质量困境的首要条件。

其次，整个团队应至力于打造合适的质量保证方法论。再次提醒一下，这里的“合适”是指“易用”和“够用”。项目团队不论资源多充足、人多聪明，都不如质量保证方法论来得实在和有效。

再次，在工作中运用三个法则。第一个法则是指：通过技术方法而不是管理方法去解决技术问题。项目开发是一个复杂的系统工程，但是其中很多问题其根源并不是来自于管理领域，而是技术领域。技术根本问题解决了，那些“管理问题”就会迎刃而解。请不要相信管理是万能的，我们必须合理地运用技术技能和管理技能。第二个法则是指：过份地强调风险其实是不能承担责任的一种表现。这一点无论是对管理者还是工程师都成立。第三个法则是指：团队的技术压力不能只源于出现技术问题时，而应在技术面前始终保持一定的压力。技术管理的最终目的是让团队将技术做好，而让团队在一定的技术压力之下有助于提升团队的技能。这三个法则作者将其分别命名为李云技术管理第一、第二和第三法则。

最后，走出质量困境离不开“坚持”两个字。无论多好的方法、多易用的工具都需要我们坚持使用。将质量做差只需一次，而要将其做好却需要永远的坚持。

26.4.1 从管理者的角度

在管理者的所有工作内容中，培养团队技能是重中之重。只有团队技能上去了，产品的质量也就随之“水涨船高”，也不容易陷入质量困境，即使身处质量困境也容易走出来。

要提高团队的技能一定要允许工程师们适当犯错（总是犯相同的错则另当别论）。改进、重构往往会带来一定的风险，而管理者如果将风险最小化作为优先考虑的因素，那么很难带领团队走出质量困境。

要提高团队的技能一定要允许团队中存在争论的声音。积极的技术争论能促进团队成员技术水平的提高，也能诱发团队成员去思考，这有利于整个团队的技能提高。

走出质量困境除了需要团队的技能外，更需要管理者对技术的敏感度。敏感度源自管理者还是工程师时的技术积累，出色的技术管理者一定曾经是出色的工程师。如果管理者不具备良好的技术敏感度，或者说团队没有具有良好技术敏感度的决策人员，那么可以肯定团队很难走出质量困境。

26.4.2　从工程师的角度

培养良好的工作习惯有助于走出（甚至远离）质量困境，因为好习惯对于软件质量也起着关键作用。软件产品的代码需要工程师一行一行地“码”出来，如果“码”代码的工程师没有良好的编程习惯，一定不能获得高质量的软件产品。

除了编程习惯外，工程师还应当培养其他的好习惯。其一是笔记习惯。只要内容有助于提高自己的工作效率和避免犯错的就应当将其记录下来。笔记不一定要记在本子上，记在计算机中也是一个很有效的方式。其二是思考习惯。对问题积极地思考以磨炼自己的洞察力，在他人提出见解时与自己的想法进行比对看看从中能学到什么。善于思考的工程师往往也爱提问和质疑。其三是阅读习惯。一个希望在技术上有所成就或做到更高层次的工程师，其知识和经验的积累一定不能只源于个人的工作，阅读是丰富技术知识很重要的一种方式。总之，养成各种好习惯的目的是为了提高自己的技能和效能。

提高自己的专业化水平同样有助于走出（甚至远离）质量困境。专业化体现在掌握能有效保证工作质量的工具和方法，以及以负责任的态度从事开发工作。质量困境的出现与我们的专业化水平不足不无关系。

26.4.3　从组织的角度

高质量软件的获得不只是开发部门和测试部门的事，从组织的层面来看，有一条质量链与软件产品如影随行。

软件开发离不开项目管理部门制定的开发计划。时间表的制定不能只考虑快速上市，还得考虑开发团队的真实技术水平。在项目管理中存在一种误区，以为时间压得越紧项目就能越早完成，而没有意识到速度的获得是以牺牲质量为代价的。安排过于紧张的时间表很容易让开发人员只顾量而不顾质。

质量管理部门在软件质量保证中也起着重要的作用。如果质量管理部门只是停留在关注软件缺陷的统计信息（只关注用户级的软件质量）上，那么可以肯定质量管理部门不会在质量链中发挥太大的作用。有效的质量管理一定需要从缺陷统计信息入手，找到不良设计的根源，并帮助开发团队通过方法论提高质量[②]。

---

② 不少公司的质量管理部门起到的是“督促”作用，而不是“帮助”作用。

无论如何，质量链上的各个部门应通力合作，用开放的心态共同致力于软件质量的提高。

26.5　小结

本章回顾了软件开发的特点，探索了软件质量保证的关键要素，以及提出了质量保证需要系统性的方法论这一观点。

质量保证方法论需要以流程和工具作为手段。流程和工具又要与开发环境做到无缝整合，以提高它们的易用性。打造质量保证方法论的其他核心手段是要素有形化和以单元测试为中心。

在最后，作者提出了走出软件质量困境的一些指导性思想。



第 27 章 编程好习惯，质量保证的基本条件

俗话说"细节决定成败"，编程习惯更多的是强调细节对软件质量的影响。笔者认为很有必要单独用一个章节来谈一谈编程好习惯。

编程习惯对于质量的影响不只反映在最终软件产品的缺陷更少这一个维度。从维护的角度来看，采用好习惯所编写的程序更具可维护性；从开发角度来看，运用好习惯所编写的程序的编译速度更快；从程序的执行效率来看，采用好习惯所编写的程序更具执行效率；等等。

编程习惯的培养需要一定的时间。所以我们需要首先从意识上认识其重要性，然后坚持不懈地打造自己所克守的、稳定的程序编写准则。培养良好的编程习惯并不只是针对嵌入式软件开发工程师的要求，而是针对所有领域的软件开发工程师。

27.1 终生受用的编程好习惯

为了方便读者的理解，对各好习惯会尽可能通过例子加以说明。

27.1.1 判断失败而非成功

图 27.1 是来自某项目简化的代码片段。

```
if (bbmt_physap_alarm_init () == RV_SUCC) {
    if (bbmt_trx_alarm_init () == RV_SUCC) {
        if (bbmt_dpd_bucket_init () == RV_SUCC) {
            if (bbmt_main_bhp_init_rfh_vars () == RV_SUCC) {
                // normal case
            }
            else {
                // error case
            }
        }
        else {
            // error case
```

```
        }
    }
    else {
        // error case
    }
}
else {
    // error case
}
```

图 27.1

这段代码看起来并不直观，如果补上每个判断语句块中的代码，有些 if 语句和与之配对的 else 离得太远，且多个 if 语句形成了嵌套，给代码阅读者带来了一定的困扰。造成这种现象的原因是，工程师在编写程序时采用了“判断成功”的策略。作者曾看到过采用这种连续嵌套的“判断成功”的代码级数多达 15 级。

下面我们换一种编程方式，改为使用“判断失败”策略。更改后的代码如图 27.2 所示。

```
if (bbmt_physap_alarm_init() != RV_SUCC) {
    // error handling
    return;
}

if (bbmt_trx_alarm_init () != RV_SUCC) {
    // error handling
    return;
}

if (bbmt_dpd_bucket_init() != RV_SUCC) {
    // error handling
    return;
}

if (bbmt_main_bhp_init_rfh_vars () != RV_SUCC) {
    // error handling
    return;
}

// normal case
```

图 27.2

更改后的代码消除了 if 嵌套语句，代码的阅读变得不再吃力，且程序只要向下走就说明不存在失败的情形。

在某些情形下，需要视具体的情况分别采用“判断成功”和“判断失败”策略。具体采用哪一种策略其依据是让代码最具可读性。

27.1.2 采用 sizeof 减少内存操作失误

在不少情形下，需要使用 memset()函数对内存进行置 0 初始化，图 27.3 中的三种错误是在

工作中经常碰到的。

```
char *buf [MAX_LEN + 1];
memset (buf, 0, MAX_LEN + 1);
```

```
#define DIGEST_LEN     17
#define DIGEST_MAX     16

char digest [DIGEST_MAX];
memset (digest, 0, DIGEST_LEN);
```

```
dll_node_t *p_node = malloc (sizeof (dll_node_t));
if (p_node == 0) {
   return;
}
memset (p_node, 0, sizeof (dll_t));
```

图 27.3

第一个错误是忘记了 buf 数组是一个字符指针数组，而不是字符数组；第二个错误是错用了宏；第三个错误出在分配时是以 dll_node_t 类型为大小，而后面 memset()时却以 dll_t 类型为大小。图 27.4 示例说明了如何采用一般的方法更正这三种错误。

```
char *buf [MAX_LEN + 1];
memset (buf, 0, sizeof (char *) * (MAX_LEN + 1));
```

```
#define DIGEST_LEN     17
#define DIGEST_MAX     16

char digest [DIGEST_MAX];
memset (digest, 0, DIGEST_MAX);
```

```
dll_node_t *p_node = malloc (sizeof (dll_node_t));
if (0 == p_node) {
   return;
}
memset (p_node, 0, sizeof (dll_node_t));
```

图 27.4

对于这类“低级”错误，我们需要思考如何在工作中尽可能避免。虽然可以通过采用代码审查的方式提高这类错误的检出率，但这样的方法并不是最好的。更为好的方法是，软件工程师通过养成良好的编程习惯去杜绝。采用 C 语言中的 sizeof()，就能显著地降低这类错误。图 27.5 示例说明了如何采用 sizeof()来更正错误。

```
char *buf [MAX_LEN + 1];
memset (buf, 0, sizeof (buf));
```

```
#define DIGEST_LEN     17
```

```
#define DIGEST_MAX     16

char digest [DIGEST_MAX];
memset (digest, 0, sizeof (digest));
```

example3.c

```
dll_node_t *p_node = malloc (sizeof (*p_node));
if (0 == p_node) {
    return;
}
memset (p_node, 0, sizeof (*p_node));
```

图 27.5

采用 sizeof()方法的共性是，以需要被初始化的目标变量名作为 sizeof()的参数。采用 sizeof()方法后，获取被初始化内存的大小可以简化为只有两条规则：

（1）如果目标变量是一个数组，则采用 sizeof（变量名）的格式获取内存大小。

（2）如果目标变量是一个指针，则采用 sizeof（*指针变量名）的格式获取内存大小。

并且采用 sizeof()并不会降低程序的性能，因为编译器在编译时就计算出结果，是一个静态行为而非运行时行为。

虽然这里是以 memset()函数为例介绍使用 sizeof()方法的，但这种方法可以运用到任何需要获取变量内存大小的场合。

27.1.3　屏蔽编程语言特性

定义数组是经常需要用到的最基本的编程语言功能之一，图 27.6 示例说明了采用数组保存一个会话 ID 的一段简化代码。

example.c

```
#define SESSION_ID_LEN_MIN     1
#define SESSION_ID_LEN_MAX     256

char g_SessionId [SESSION_ID_LEN_MAX];

int save_session_id (char *_session_id, int _length)
{
    if (_length < SESSION_ID_LEN_MIN || _length > SESSION_ID_LEN_MAX) {
        return ERROR;
    }

    memcpy (g_SessionId, _session_id, _length);
    g_SessionId [_length] = '\0';

    return SUCCESS;
}
```

图 27.6

如果仔细观察将能发现图 27.6 中存在的一个缺陷，这个缺陷是当_length 参数值的大小刚

好是 SESSION_ID_LEN_MAX 时，就会造成数组写越界。为了修复这一缺陷，可能会采用图 27.7 所示的方法。

```
example.c
#define SESSION_ID_LEN_MIN    1
#define SESSION_ID_LEN_MAX    256

char g_SessionId [SESSION_ID_LEN_MAX];

int save_session_id (char *_session_id, int _length)
{
   if (_length < SESSION_ID_LEN_MIN || _length >= SESSION_ID_LEN_MAX) {
      return ERROR;
   }

   memcpy (g_SessionId, _session_id, _length);
   g_SessionId [_length] = '\0';

   return SUCCESS;
}
```

图 27.7

其中改动的出发点是判断当_length 的大小为 SESSION_ID_LEN_MAX 时，让程序返回错误。从功能上来说，这一改动没有任何问题，但其中存在一个可维护性问题，这个问题就是由改动后的“>=”造成的。

先抛开编程语言的语法，如果某个数学变量的最大值是 Y，那么 Y 是这个变量的有效取值吗？作者的理解是：这个值应当是变量的有效取值。现在回头看一看前面的改动，其中将最大值当做一个无效的取值，这显然违背了通常意义上对最大值的理解。

我们应力争所编写程序的语义不会与常识相悖。对于上面更改后的代码，当程序的阅读者读到“>=”时，很可能会停下来思考一下“为什么不能等于最大值呢？”可以想象，阅读者得查看一下 g_SessionId 数组的大小是多少，然后“哦，这是因为数组的大小是 SESSION_ID_LEN_MAX，而数组大小的定义将字符串中的结束符'\0'也计算在内，所以_length 的大小不能等于 SESSION_ID_LEN_MAX”。

使用“>=”方法修订程序所带来的问题是，将通常的公共语言与编程语言糅在一起，从而造成所编写的程序不容易理解。那 SESSION_ID_LEN_MAX 的定义应当包括最后的那个结束符吗？不应该，因为字符串最后的结束符是从 C 语言的角度去看而存在的。图 27.8 是推荐的更好方法。

```
example.c
#define SESSION_ID_LEN_MIN    1
#define SESSION_ID_LEN_MAX    255

char g_SessionId [SESSION_ID_LEN_MAX + 1];

int save_session_id (char *_session_id, int _length)
```

```
{
    if (_length < SESSION_ID_LEN_MIN || _length > SESSION_ID_LEN_MAX) {
        return ERROR;
    }

    memcpy (g_SessionId, _session_id, _length);
    g_SessionId [_length] = '\0';

    return SUCCESS;
}
```

图 27.8

这一改动非常简单，就是在定义 g_SessionId 数组时，在其大小后面增加一个“+1”，用于消化掉 C 语言中的字符串应以'\0'结束这一特性。另外，让 SESSION_ID_LEN_MAX 减一，以表示不包含字符串中的结束符。这种方法的好处是：

（1）没有违背大家对最大值的理解，即最大值是一个可取的有效值。

（2）可以采用一致的语义来编写程序，后继的维护人员可以很容易地理解，也就不容易出错了。

图 27.9 是另一个存在混合编程语言特性和公共语言的程序，其中的 section_array_数组的大小被定义为 MAX_SECTION_COUNT，而在 section_add() 函数中，将 _id 参数与 MAX_SECTION_COUNT 进行比较以防止出现数组越界。显然，MAX_SECTION_COUNT 从字面意思上理解是可以取值的，但由于_id 是从 0 开始的，于是就出现了_id 不能等于 MAX_SECTION_COUNT 这一情形。

```
const int MAX_SECTION_COUNT = 64;

msection_t section_array_[MAX_SECTION_COUNT];

void section_add (section_id_t _id, addr_t _start, size_in_page_t _size)
{
    if (MAX_SECTION_COUNT <= _id) {
        assert (false);
    }
    section_array_[_id].attach (_start, _size);
}
```

图 27.9

图 27.10 实现了增加另一个定义从而达到屏蔽编程语言的特性这一目的。MAX_SECTION_ID 被定义为 MAX_SECTION_COUNT 减一，于是在 section_add()函数中，对于_id 变量值的判断就变成去掉了等于号。

```
const int MAX_SECTION_COUNT = 64;
const int MAX_SECTION_ID = MAX_SECTION_COUNT - 1;
msection_t section_array_[MAX_SECTION_COUNT];
```

```
void section_add (section_id_t _id, addr_t _start, size_in_page_t _size)
{
    if (MAX_SECTION_ID < _id) {
        assert (false);
    }
    section_array_[_id].attach (_start, _size);
}
```

图 27.10

这一编程习惯所给出的启示是：软件工程师在编写程序时，应当尽可能站在一个不懂编程语言特性的角度去思考。这种方式就是要求我们不要将不同的概念混在一起，以免造成可维护性问题。

27.1.4　恰当使用 goto 语句

goto 语句可谓是“臭名昭著”，乃至有的书或公司的编程规范提出禁用 goto 语句的说法，结果造成有的软件工程师一看到 goto 语句在某程序中被使用，就本能地认为这个程序写得很“垃圾”。但是，凡事都不能太偏激，goto 语句运用得好能大大地简化程序，以及提高程序的可读性和可维护性。

在开始示例说明其好处之前，先用一些统计数据来说明 goto 语句并没有因为“臭名昭著”而被抛弃，这些统计数据可能并不是百分之百的精确，但很具有说服力。对于操作系统，Linux-2.6.21 内核使用了 20333 个、VxWorks-6.2 使用了 9142 个、941 个被运用到了 rtems-4.9.2 中，glibc-2.9 库则使用了 1750 个。所有的这些统计数据都表明，goto 语句并没有遭到禁用。

图 27.11 是一个没有采用 goto 语句编写的函数。其中存在多处出错处理的代码，比如第 112～113 行、第 118～119 行和第 123～126 行。采用这种分布式的出错处理很容易出现遗漏，造成忘记释放资源而产生问题。

```
00097: int queue_init (queue_t ** _pp_queue, int _size)
00098: {
00099:     pthread_mutexattr_t attr;
00100:     queue_t *queue;
00101:
00102:     queue = (queue_t *) malloc(sizeof(queue_t));
00103:     if (0 == queue) {
00104:         return -1;
00105:     }
00106:     *_pp_queue = queue;
00107:
00108:     memset (queue, 0, sizeof (*queue));
00109:     queue->size_ = _size;
00110:     pthread_mutexattr_init (&attr);
00111:     if (0 != pthread_mutex_init(&queue->mutex_, &attr)) {
00112:         pthread_mutexattr_destroy (&attr);
00113:         free (queue);
00114:         return -1;
00115:     }
00116:     queue->messages_ = (void **) malloc (queue->size_ * sizeof (void *));
```

```
00117:     if (0 == queue->messages_) {
00118:         pthread_mutexattr_destroy (&attr);
00119:         free (queue);
00120:         return -1;
00121:     }
00122:     if (0 != sem_init(&queue->sem_put_, 0, queue->size_)) {
00123:         free (queue->messages_);
00124:         pthread_mutexattr_destroy (&attr);
00125:         free (queue);
00126:         return -1;
00127:     }
00128:     pthread_mutexattr_destroy (&attr);
00134:     return 0;
00135: }
```

图 27.11

图 27.12 是采用 goto 语句所编写的另一个版本，与不采用 goto 语句的版本相比，程序更加简洁，在出错处理的地方大多使用 goto 语句跳转到程序的末尾集中进行错误处理。

```
example.c
00053: int queue_init (queue_t ** _pp_queue, int _size)
00054: {
00055:     pthread_mutexattr_t attr;
00056:     queue_t *queue;
00057:
00058:     queue = (queue_t *) malloc(sizeof(queue_t));
00059:     if (0 == queue) {
00060:         return -1;
00061:     }
00062:     *_pp_queue = queue;
00063:
00064:     memset (queue, 0, sizeof (*queue));
00065:     queue->size_ = _size;
00066:     pthread_mutexattr_init (&attr);
00067:     if (0 != pthread_mutex_init(&queue->mutex_, &attr)) {
00068:         goto error;
00069:     }
00070:     queue->messages_ = (void **) malloc (queue->size_ * sizeof (void *));
00071:     if (0 == queue->messages_) {
00072:         goto error;
00073:     }
00074:     if (0 != sem_init(&queue->sem_put_, 0, queue->size_)) {
00075:         goto error1;
00076:     }
00077:     pthread_mutexattr_destroy (&attr);
00078:     return 0;
00079:
00080: error1:
00081:     free (queue->messages_);
00082: error:
00083:     pthread_mutexattr_destroy (&attr);
00084:     free (queue);
00085:     return -1;
00086: }
```

图 27.12

goto 语句除了可以用在这里所示例说明的出错处理中外，还可以用在其他的程序逻辑中以

提高代码的可读性。使用 goto 语句时需要注意以下原则。

（1）不能滥用。

（2）不要让 goto 语句形成一个环。使用 goto 语句应形成一条线，从一点跳到另一点。如果 goto 语句的使用没有破坏可读性，那么可以适当地考虑打破这一原则。

27.1.5　合理运用数组

对于多任务或多线程环境的程序，不少任务的生命周期与整个程序的生命周期是一致的，即它们是在程序初始化时创建的，然后运行到程序结束。对于这种任务后面称它具有全局生命周期。

如果具有全局生命周期的任务需要内存资源，我们完全可以用定义全局或静态数组的方式来代替动态分配的方式。图 27.13 示例说明了 thread_authenticator 线程采用 malloc()函数初始化它的全局变量 g_aaa_eap_str_buff 的代码，图 27.14 则示例说明了采用数组的方式。

```
#define MAX_AAA_SS_PORTS          64
#define MAX_NUM_RADIUS_IDS        (MAX_AAA_SS_PORTS * 256)
#define MAX_EAP_MESSAGE_LEN       4096

static char **g_aaa_eap_str_buff;

void thread_authenticator (void *_arg)
{
   g_aaa_eap_str_buff = (char **) malloc (MAX_NUM_RADIUS_IDS);
   if (0 == g_aaa_eap_str_buff) {
      log_error ("Failed to allocate buffer for storing eap strings");
      return;
   }

   for (int i = 0; i < MAX_NUM_RADIUS_IDS; i ++) {
      g_aaa_eap_str_buff [i] = (char *) malloc (MAX_EAP_MESSAGE_LEN);
      if (0 == g_aaa_eap_str_buff [i]) {
         log_error ("Failed to allocate buffer for storing eap strings");
      }
   }

   while (1) {
      ……
   }
}
```

图 27.13

```
#define MAX_AAA_SS_PORTS          64
#define MAX_NUM_RADIUS_IDS        (MAX_AAA_SS_PORTS * 256)
#define MAX_EAP_MESSAGE_LEN       4096

char g_aaa_eap_str_buff [MAX_NUM_RADIUS_IDS][ MAX_EAP_MESSAGE_LEN];
```

```
void thread_authenticator (void *_arg)
{
    while (1) {
        ......
    }
}
```

图 27.14

很明显，采用数组获取内存的方式所编写的代码更简洁，因为省去了对 malloc()函数的调用；而且内存的释放无须我们控制，这也就降低了内存泄漏的可能性。

但采用数组获取内存的方式有一点需要注意：由于全局或静态数组一旦定义，它所占用的内存在运行期间就不能被释放，因此在使用数组这种方式预留内存时，需要注意是否会带来内存浪费问题。

27.1.6 以逆序方式释放资源

软件的功能似乎都是以资源的管理为主线的，各模块在实现时或多或少存在对资源的分配和释放。一个模块在通过分配获取所需资源时，无一例外地存在先后顺序。在图 27.15 的 queue_init()函数中，在第 58、68、72、77 和 81 行分别进行了资源分配操作。当一个队列不再需要时，需要调用 queue_fini()函数释放在 queue_init()函数中分配所获得的资源。

```
00053: int queue_init (queue_t ** _pp_queue, int _size)
00054: {
00055:     pthread_mutexattr_t attr;
00056:     queue_t *queue;
00057:
00058:     queue = (queue_t *) malloc(sizeof(queue_t));
00059:     if (0 == queue) {
00060:         return -1;
00061:     }
00062:     *_pp_queue = queue;
00063:
00064:     memset (queue, 0, sizeof (*queue));
00065:     queue->size_ = _size;
00066:
00067:     pthread_mutexattr_init (&attr);
00068:     if (0 != pthread_mutex_init(&queue->mutex_, &attr)) {
00069:         goto error1;
00070:     }
00071:
00072:     queue->messages_ = (void **) malloc (queue->size_ * sizeof (void *));
00073:     if (0 == queue->messages_) {
00074:         goto error1;
00075:     }
00076:
00077:     if (0 != sem_init(&queue->sem_put_, 0, queue->size_)) {
00078:         goto error2;
00079:     }
00080:
00081:     if (0 != sem_init(&queue->sem_get_, 0, 0)) {
00082:         goto error3;
```

```
00083:     }
00084:
00085:     pthread_mutexattr_destroy (&attr);
00086:
00087:     return 0;
00088:
00089: error3:
00090:     sem_destroy (&queue->sem_put_);
00091: error2:
00092:     free (queue->messages_);
00093: error1:
00094:     pthread_mutexattr_destroy (&attr);
00095:     free (queue);
00096:     return -1;
00097: }
00098:
00099: int queue_fini (queue_t ** _pp_queue, int _size)
00100: {
00101:     queue_t *queue = *_pp_queue;
00102:
00103:     if (0 == queue) {
00104:         return -1;
00105:     }
00106:
00107:     sem_destroy (&queue->sem_get_);
00108:     sem_destroy (&queue->sem_put_);
00109:     free (queue->messages_);
00110:     pthread_mutex_destroy (&queue->mutex_);
00111:     free (queue);
00112: }
```

图 27.15

queue_fini()在释放资源时，应当以逆序的方式释放分配所获得的资源，即使资源释放的顺序根本不重要。图中的第 107 和 108 行的顺序就不重要，但第 109 和 111 行的顺序就不能颠倒，否则会造成软件崩溃。

将以逆序的方式释放资源作为编程习惯，有助于避免因资源释放顺序不对所造成的问题。

27.1.7　在模块对外接口中防范错误

作为自我保护的一个编程好习惯，应在模块的对外接口函数中防范潜在的参数传入错误。如果一个函数的输入参数是一个不能为 NULL 的指针，且这一函数作为一个模块的对外接口，那一般情况下我们需要在函数的开始处检查指针是否为 NULL，这种做法就是 C 语言中所说的“不变式”，即检查函数的参数没有违背前提假设。

不是每一个函数都要检查输入参数的有效性。如果函数是在模块内部使用的，则可以考虑省去对参数有效性的检查，或者使用 C 库中的 assert()函数就够了。不少程序因为不区分模块接口函数和内部函数，盲目地检查所有函数参数的有效性，造成程序代码看上去很拖沓。

有些函数为了效率并不检查参数的有效性，比如很多 C 语言的库函数，而是将参数的有效性保证交给了使用者。本书代码中所使用到的双向链表模块（dll.c）中的所有函数就没有进行

参数的有效性检查。

27.1.8 避免出现魔数

魔数（magic number），即在编写程序时直接在程序中运用数字，而不是采用定义宏或者const 变量的方式。图 27.16 是使用了魔数的一个示例程序，其中的 64 是指 Msk 的最大字节数。

```
00290: #define MIN_MSK_LEN     20
00291:
00292: int adjustMsk (MskContext* Context)
00293: {
00294:     char temp [64] = {0};
00295:
00296:     if (Context->lenMsk > 64) {
00297:         memcpy (temp, Context->msk + (Context->lenMsk - 64),    64);
00298:         ……
00399:         memcpy (Context->msk, temp, 64);
00300:     }
00301:     else if (Context->lenMsk < MIN_MSK_LEN) {
00302:         return ERROR;
00303:     }
00304:     ……
00305: }
```

图 27.16

采用魔数的危害有：

（1）降低了程序的可读性。有人可能会想到增加注释的解决方法。如果真是采用加注释的方式，那为什么不将其定义成一个宏或者 const 常量呢？要知道查看注释的效率肯定没有直接看代码来得快和方便，也不存在代码与注释不同步的问题。

（2）如果下一次这个最大值要从 64 改为 128，那得在 adjustMsk()中对每一处都进行更改。

由此看来，这里的“魔”不应理解成像“魔法（magic）”那样神奇，而应理解为像“魔鬼（monster）”那样可怕。

图 27.17 是引入宏之后的版本。其中定义了 MAX_MSK_LEN 的大小为 64，如果其他函数中也需要用到 Msk 的最大值，那么也可以引用这个宏。如果下一次想将最大值从 64 改为 128 时，只要改 MAX_MSK_LEN 宏的定义就行了。另外，这种宏定义的存在有利于模块与模块之间共享，从而在一定程度上提高重用性。

```
00289: #define MIN_MSK_LEN     20
00290: #define MAX_MSK_LEN     64
00291:
00292: int adjustMsk (MskContext* Context)
00293: {
00294:     char temp [MAX_MSK_LEN] = {0};
00295:
```

```
00296:     if (Context->lenMsk > MAX_MSK_LEN) {
00297:         memcpy (temp, Context->msk + (Context->lenMsk - MAX_MSK_LEN), sizeof (temp));
00398:         ......
00399:         memcpy (Context->msk, temp, sizeof (temp));
00300:     }
00301:     else if (Context->lenMsk < MIN_MSK_LEN) {
00302:         return ERROR;
00303:     }
00304:     ......
00305: }
```

图 27.17

27.1.9　利用编程语言特性提高效率

利用编程语言特性不但能简化程序，而且还能提高程序的执行效率。先看一个使用 sizeof() 提高程序效率的例子。图 27.18 是没有使用 sizeof()之前的代码，其背景需要在此做一个交代。其中，alarm_string 变量是定义为长度是 255 的字符数组，而 tail_msg 定义的是一个指向字符串“, List NOT Complete”的指针。space 变量是为了得到在 alarm_string 中除去 tail_msg 所指向字符串的长度后，有多少空间可以用来存放其他的内容。另外，tail_msg 所指向的字符串内容是不会被更改的。

```
example.c
00070: #define MAX_STRING_TXT        255
00071: char alarm_string [MAX_STRING_TXT];
00072:
00073: char *tail_msg = ", List NOT Complete";
00074: int space = MAX_STRING_TXT - strlen (tail_msg) - 1;
```

图 27.18

在图 27.18 中，为了计算 space 的值需要用到 strlen()函数以得到 tail_msg 所指向字符串的长度。由于 strlen()并不将字符串的结束符'\0'计算在内，所以 space 的最后面还得减 1。再由于 strlen()是一个函数，所以这段代码在被执行时将耗费一定的处理器时间。

图 27.19 是更高效的一种实现。其中将 tail_msg 定义为一个静态数组，且在 space 变量的计算中使用 sizeof()进行替代。注意，sizeof()会将字符串的结束符'\0'计算在内。

```
example.c
00070: #define MAX_STRING_TXT        255
00071: char alarm_string [MAX_STRING_TXT];
00072:
00073: static const char tail_msg [] = ", List NOT Complete";
00074: int space = MAX_STRING_TXT - sizeof (tail_msg);
```

图 27.19

sizeof()的值在编译时就决定了。对于这里的例子，编译器在编译时就会计算出 sizeof（tail_msg）的值应当是 20，因此 space 在运行时将会被直接赋值为 235，而不存在任何的运行时函数调用和数学运算。另外，还需要注意 tail_msg 应当定义为 static 和 const，否则编译器会

生成一段代码，当每次程序被执行时都会对位于栈上的 tail_msg 数组进行初始化。将 tail_msg 定义为 static 和 const 就会造成其内存分配在.rodata 段（参见 9.1 节）上而不是栈上，从而避免多次进行初始化操作。

下面再看另一个使用编程语言特性的例子。图 27.20 的第 143 行调用 memset()对局部数组变量 temp 进行置 0 初始化[①]，这段代码每次运行时都得进行 memset()函数调用。

```
00141: #define MAX_MSK_OCTET_LEN        64
00142: char temp[MAX_MSK_OCTET_LEN];
00143: memset (temp, 0, sizeof(temp));
```

图 27.20

图 27.21 是更优的实现。其中的改动是在 temp 变量的最后加上一个初始化为 0 的赋值，当编译器看到这段代码时，会生成代码对 temp 所指向的全部内存（即 64 个字节）进行置 0 初始化。如此一来，就省去了对 memset()函数的调用，从而达到提高效率的目的。

```
00141: #define MAX_MSK_OCTET_LEN        64
00142: char temp[MAX_MSK_OCTET_LEN] = {0};
```

图 27.21

要运用好编程语言的特性，需要对编程语言有深入的理解，而不能只局限于一些入门书籍中所介绍的知识。尽管运用编程语言特性所带来的效率提高对于现在强大的处理器而言可以忽略不计，但这能体现我们的专业性——对于编程语言的娴熟驾驭！

27.1.10　复用代码提高维护性

代码复用在软件开发中存在两个层次。一个层次是，在设计一个新的软件功能或者开发一个新的项目时，复用已存在的软件模块，这种复用或许称为设计复用更好；另一个层次是，软件工程师在开发一个软件模块时，模块的内部应尽可能地实现函数复用。本节所谈及的复用指的是后者。

现在假设存在一个双向链表（Double-Linked List，DLL）的模块，如果这个模块在开发的某时刻已存在两个函数，分别是 dll_push_tail()和 dll_pop_head()，这两个函数的作用分别是将一个新的节点加入到链表的尾部，以及从链表中删除并返回头节点。其程序实现如图 27.22 所示。

```
00088: void dll_push_tail (dll_t *_p_dll, dll_node_t *_p_node)
00089: {
00090:     if (0 ==_p_dll->tail_) {
00091:         _p_dll->head_ = _p_dll->tail_ = _p_node;
```

① 这里假设 temp 是局部变量，而不是全局变量。

```
00092:         _p_node->next_ = _p_node->prev_ = 0;
00093:     }
00094:     else {
00095:         dll_node_t *p_tail = _p_dll->tail_;
00096:
00097:         p_tail->next_ = _p_node;
00098:         _p_node->prev_ = p_tail;
00099:         _p_node->next_ = 0;
00100:         _p_dll->tail_ = _p_node;
00101:     }
00102:
00103:     _p_dll->count_ ++;
00104: }
00105:
00106: dll_node_t *dll_pop_head (dll_t *_p_dll)
00107: {
00108:     dll_node_t *p_node = _p_dll->head_;
00109:
00110:     if (p_node != 0) {
00111:         _p_dll->count_--;
00112:         _p_dll->head_ = p_node->next_;
00113:         if (0 ==_p_dll->head_) {
00114:             _p_dll->tail_ = 0;
00115:         }
00116:         else {
00117:             p_node->next_->prev_ = 0;
00118:         }
00119:     }
00120:
00121:     return p_node;
00122: }
```

图 27.22

如果此时需要增加一个新的链表操作函数 dll_merge()用于合并两个链表，则这个函数的实现可能如图 27.23 所示。思路很简单，就是从_p_src 链表中将一个个的节点取出并放到_p_dest 链表的尾部。

```
00165: void dll_merge (dll_t *_p_dest, dll_t *_p_src)
00166: {
00167:     dll_node_t *p_node = _p_src->head_;
00168:
00169:     while (0 != p_node) {
00170:         if (0 ==_p_dest->tail_) {
00171:             _p_dest->head_ = _p_dest->tail_ = p_node;
00172:             _p_dest->next_ = _p_dest->prev_ = 0;
00173:         }
00174:         else {
00175:             dll_node_t *p_tail = _p_dest->tail_;
00176:
00177:             p_tail->next_ = _p_dest;
00178:             _p_dest->prev_ = p_tail;
00179:             _p_dest->next_ = 0;
00180:             _p_dest->tail_ = _p_dest;
00181:         }
00182:
00183:         _p_dest->count_ ++;
```

```
00184:         p_node = p_node->next_;
00185:     }
00186:
00187:     _p_src->count_ = 0;
00188:     _p_src->head_ = 0;
00189:     _p_src->tail_ = 0;
00190: }
```

图 27.23

从功能性的角度来说没有问题，但是从可维护性方面来看，这一实现并不好，取而代之的更好实现是通过代码复用的方式，如图 27.24 所示。

```
dll.c
00175: void dll_merge (dll_t *_p_dest, dll_t *_p_src)
00176: {
00177:     dll_node_t *p_node = dll_pop_head (_p_src);
00178:
00179:     while (0 != p_node) {
00180:         dll_push_tail (_p_dest, p_node);
00181:         p_node = dll_pop_head (_p_src);
00182:     }
00183: }
```

图 27.24

很明显，采用代码复用的方式所编写出的 dll_merge()函数更具可读性。在实现一个软件模块时，应当考虑从所需实现的功能中抽取出一些公共的基本函数（比如，这里谈到的 dll_pop_head()和 dll_push_tail()），且这些函数所实现的功能是正交的（即功能没有重叠）。接下来，其他的功能（比如这里谈到的 dll_merge()）就可以考虑采用搭积木的方式，通过运用那些最基本的函数去实现。

采用复用方式实现的 dll_merge()引入了函数调用，而函数调用因为存在参数的传递可能会带来一定的处理器开销，其开销的大小与处理器的处理能力有关。但是，对于现代的大多处理器来说，这种开销都是很小的。在性能不是问题的前提下，我们应尽可能保证代码的可维护性。

27.1.11 借助隐式初始化简化程序逻辑

在图 27.25 的开始处示例说明了三个以“mprotector_”开头的函数原型。假设 mprotector_section_add()函数将会被多个任务调用，以便用于初始化各任务相关的数据，但在调用它之前必须保证 mprotector_init()函数被执行过且只能执行一次。

```
int mprotector_init ();
int mprotector_fini ();
int mprotector_section_add (section_id_t _id, maddr_t _start, msize_t _size);

void task_entry_A (void *_p_arg)
{
   maddr_t start;
   msize_t size;
```

```
    ......
    mprotector_section_add (SECTION_1, start, size);
    ......
}

void task_entry_B (void *_p_arg)
{
    maddr_t start;
    msize_t size;
    ......
    mprotector_section_add (SECTION_2, start, size);
    ......
}
```

图 27.25

图中也示例说明了任务 A 和 B 分别在它的体函数内调用 mprotector_section_add()。那 mprotector_init()应当放在什么地方被调用呢？放到任务 A 和 B 的入口函数内不太好，这会造成一旦任务的初始化先后顺序更改后结果会随之改变。很容易想到的另一种方法是，在创建任务 A 和 B 之前，先调用 mprotector_init()。

更好的方法是采用隐式初始化的方法，这得对 mprotector_init()和 mprotector_section_add()两个函数的实现做一些更改，如图 27.26 所示。mprotector_init()中的更改是允许它被多次调用，当发现是第二次被调用时立即返回。在 mprotector_section_add()中则增加对 mprotector_init()的调用。

```
int mprotector_init ()
{
    static bool initialized = false;

    if (initialized) {
        return 0;
    }
    ......
    initialized = true;
    return 0;
}

int mprotector_section_add (section_id_t _id, maddr_t _start, msize_t _size)
{
    if (0 != mprotector_init ()) {
        return -1;
    }
    ......
}
```

图 27.26

有了这些更改以后，使用 mprotector_section_add()函数的用户就根本不需要考虑在什么地方调用 mprotector_init()函数，这也就简化了程序逻辑。

除了简化逻辑，采用隐式初始化这一方法能有效地解决模块间复杂的依赖关系。

27.1.12　青睐小粒度锁

对于多任务程序，经常需要用到互斥锁或开关中断的方式以防止竞争问题。编程的另一个好习惯是，让上锁的粒度尽可能小，这有助于提高系统的实时性和性能。

对于图 27.27 中的代码，timer_init()函数在对_handle 数据结构进行初始化之前，在第 123 行先进行上锁操作，在完成了数据结构的初始化之后，则在第 145 行进行解锁操作。这段代码在功能上没有问题，但是它没有做到让上锁的粒度尽可能小。

```
00096: typedef struct tag_timer {
00097:     dll_node_t node_;
00106:     ......
00107: } timer_instance_t, *timer_handle_t;
00108:
00109: // guard for initialized timer
00110: static const csize_t TIMER_MARK = 0x20091026;
00111:
00112: error_t timer_init (timer_handle_t _handle, msecond_t _duration,
00113:     expiry_cb_t _cb, const char *_name)
00114: {
00122:     ......
00123:     timer_lock ();
00124:     if (TIMER_MARK == _handle->mark_) {
00125:         // if it has been initialized then failing it
00126:         timer_unlock ();
00127:         return ERROR_T (ERROR_TIMER_INIT_INITIALIZED);
00128:     }
00129:
00130:     // convert to TICK_DURATION_IN_MSEC unit
00131:     _handle->ticks_ = _duration / TICK_DURATION_IN_MSEC;
00132:     if (0 == _handle->ticks_) {
00133:         _handle->ticks_ ++;
00134:     }
00135:     _handle->cb_ = _cb;
00136:     _handle->state_ = TIMER_INITIALIZED;
00137:     dll_node_init (&_handle->node_);
00138:     if (0 == _name) {
00139:         _handle->name_ [0] = 0;
00140:     }
00141:     else {
00142:         strncpy (_handle->name_, _name, sizeof (_handle->name_));
00143:     }
00144:     _handle->mark_ = TIMER_MARK;
00145:     timer_unlock ();
00146:
00147:     return 0;
00148: }
```

图 27.27

如果仔细阅读这段代码，读者会发现在第 124 行会检查 mark_变量是否已被设置为 TIMER_MARK 所定义的值，如果已设置则进行出错处理，其逻辑依据就是一个已初始化的定时器不能被再一次初始化。继续往下看的话，读者能看到在第 144 行会对 mark_变量进行设值操作。

为了减小上锁的粒度，只要将图 27.27 中的第 114～115 行向上移就行了，更改后的代码如图 27.28 所示，短短的第 124～130 行就能保证上锁的时间最短，且不失对竞争问题的避免。

```
00112: error_t timer_init (timer_handle_t _handle, msecond_t _duration,
00113:     expiry_cb_t _cb, const char *_name)
00114: {
00122:     ......
00123:     timer_lock ();
00124:     if (TIMER_MARK == _handle->mark_) {
00125:         // if it has been initialized then failing it
00126:         timer_unlock ();
00127:         return ERROR_T (ERROR_TIMER_INIT_INITIALIZED);
00128:     }
00129:     _handle->mark_ = TIMER_MARK;
00130:     timer_unlock ();
00122:     ......
00148: }
```

图 27.28

timer_init()函数中所展现的是锁的时间维度的粒度，除此之外，还有资源维度的粒度。如果一个模块需要 A 和 B 两个不同的独立资源，且这一模块中的有些函数只需用到 A 资源，或者有的函数只需用到 B 资源。在 A 和 B 都需要运用锁进行保护的情形下，应当为 A 和 B 设计两个不同的锁而不是同一个，通过“专锁专用”来减小锁所控制资源的粒度。

27.1.13　精确包含头文件

请注意这里用的是“精确”而不是“正确”，之所以不说“正确”，是因为如果头文件没有被正确包含的话，编译器是不会生成最终的目标代码的。那用“精确”一词想表达除“正确”之外的什么意思呢？

“精确”一词想表达的第一层意思是，只包含必需的头文件。图 27.29 是一个简单的示例程序。

```
#include <stdio.h>
#include <time.h>

void foo ()
{
    printf ("Just for an example!\n");
}
```

图 27.29

我们知道 printf()函数的原型声明来源于标准库的 stdio.h 头文件，foo.c 文件中除了包含 stdio.h 头文件外还包含了另外一个多余的头文件——time.h，从编译结果来看没有问题。

但是当一个项目很大时，这种类似的行为会凸显编译效率低下这一问题。当 foo.c 文件被

编译时，第一步要做的是预处理，预处理的最终结果可以看做是将 stdio.h 和 time.h 中的内容全部放到 foo.c 文件中。如果 stdio.h 和 time.h 中又包含其他的头文件的话，它们也都会全部被放入到最终的文件中。图 27.30 示例说明了采用 gcc 的-E 选项所获得的最终预处理完的文件 final.c。

```
$ gcc -E foo.c > final.c


$ vi final.c
  1414 # 3 "foo.c" 2
  1415
  1416 void foo ()
  1417 {
  1418     printf ("Just for a test!\n");
  1419 }
:set nu
```

图 27.30

从图中可以看出，本来是 7 行的文件变成了 1419 行，这意味着如果多包含头文件，则编译所花费的时间将更长。因为，编译器在最终分析 final.c 文件中的词法和语法时，它必须从头到尾一行一行地处理。千万别小看对于每一个文件多出来的那么一点时间，当一个项目有很多文件时，所多出来的时间就不可小视了。

当因为没有“精确”包含头文件使得项目的编译效率成为一个不可忽视的问题时，要消除它所需付出的努力已经很大了，甚至是项目团队不可承受的，且这种纠正行为几乎是“体力劳动”。

“精确”一词想表达的第二层意思是，尽可能不要在头文件中包含其他的头文件，取而代之的是尽量在.c 源程序文件中包含它们。对于图 27.31 和图 27.32 两种实现方式，从 foo()函数的角度来说是一模一样的，只是对于 stdio.h 头文件的包含一个是放在 foo.h 中的，而另一个则是放在 foo.c 中的。

```
#include <stdio.h>

void foo ();
```

```
#include "foo.h"

void foo ()
{
   printf ("Just for an example!\n");
}
```

图 27.31

```
foo.h
void foo ();
foo.c
#include <stdio.h>
#include "foo.h"

void foo ()
{
    printf ("Just for an example!\n");
}
```

图 27.32

在大部分情形下，设计一个模块的目的是需要供其他模块使用，而其他模块在需要使用 foo() 函数时通常需要包含 foo.h 头文件。采用图 27.31 的实现方式将造成包含 foo.h 头文件会导致间接地包含 stdio.h，进而导致前面所讲的编译速度下降问题。而图 27.32 的实现就不存在这一问题。printf()函数只是在 foo()函数的实现中需要被使用，而完全可以不让 foo.h 头文件知道这一信息，所以根本就没有必要在 foo.h 头文件中包含 stdio.h。

相信不少读者有这种类似的项目经验，即为了包含头文件省事，通过定义一个包罗万象的公共头文件，然后让其他的源程序文件只需包含这个头文件就行了。相信读者现在明白了这并不是一种好实践。

27.1.14　让模块的对外头文件保持简洁

从“精确包含头文件”这一编程好习惯中可以得到另外一个好习惯，那就是尽可能让模块的对外头文件保持简洁。

通常的编程习惯是将所有的数据结构都放在头文件中，且这个头文件也包含模块接口函数的原型声明，图 27.33 就是一个例子。请注意其中的第 62～69 行所定义的 bucket_t 结构，这一结构是模块的内部数据结构，也就是说，从模块的外部来看根本不需要知道这一结构的存在，这从第 76～83 行模块所有函数的原型声明中可以看出，因为这些函数的参数和返回类型中都没有引用它。

```
00037: typedef struct tag_timer *timer_handle_t;
00038:
00039: // callback function for timer expiration
00040: typedef void (*expiry_cb_t)(timer_handle_t _handle, void *_arg);
00049:
00050: typedef struct tag_timer {
00051:     ......
00060: } timer_instance_t;
00061:
00062: typedef struct {
00063:     ......
00069: } bucket_t;
00070:
00076: int timer_lock_init ();
```

```
00077: void timer_fire ();
00078: error_t timer_alloc (timer_handle_t *_p_handle, msecond_t _duration,
00079:     expiry_cb_t _cb, const char *_name);
00080: error_t timer_free (timer_handle_t _handle);
00081: error_t timer_start (timer_handle_t _handle, void *_arg);
00082: error_t timer_stop (timer_handle_t _handle);
00083: void timer_dump ();
```

图 27.33

bucket_t 结构的定义放在 timer.h 头文件中合适吗？从简化模块头文件的角度来说是不合适的，因为外部在使用定时器模块时并不需要使用到 bucket_t 结构，因此应当将它从 timer.h 中移出去。存在两种做法：或者将这一结构放入 timer.c 文件中，或者定义一个只用来被 timer.c 包含的私有头文件。当内部数据结构只需要被一个.c 文件引用时，前一种方法更简单，这可以省去新增加一个文件的麻烦；但是，当有多个.c 文件需要使用它时，则第二种方法才可行。

简化模块头文件的目的就是为了让其他模块的.c 文件在包含它时，可以做到所需编译的文件大小最小化，从而提高程序的编译效率。

27.1.15　只暴露必要的变量和函数

在设计一个模块时，应做到避免暴露只在模块内部使用的变量和函数，这可以通过使用 static 关键字做到。

如果一个内部变量或函数并没有被声明成 static，那意味着所设计的模块存在“洞”。通过这种“洞”其他的模块可以窥视到模块的内部实现，或通过这些“洞”影响模块的内部行为。

理论上，一个模块不应当暴露任何一个内部变量，除非因为不可避免的某种因素，否则都应当通过提供接口函数的形式实现对内部变量的存取。通过提供接口函数这种方式的好处是，如果某一天模块的实现需要更改变量的话，完全可以让模块的使用者感知不到这种变化。另外，将变量定义为 static，可以避免在某种情形下因为各模块无意间重复定义同样的名字而导致“离奇”的软件缺陷，11.3 节解释了为什么会有这类问题发生。

如果一个函数只在模块内部使用的话，其原型声明也不应当出现在模块的头文件中。出现在头文件中的函数原型，其所隐含的意思就是这些函数是外部模块可调用的。

定义为 static 的变量和函数，可以被他人轻松地去掉 static 关键字而突破原本希望的限制。但无论如何，static 的存在都是给试图这样做的人一个小小的警告——“这是 static 的，你真的想去掉而打破原先的约束吗？”这一警告对于专业的软件工程师来说足以引发他的思考。

27.1.16　清除编译器报告的所有警告

不少软件缺陷是因为编译器报告警告而我们并没有认真对待而导致的。清除编译器所发出的每一个警告也是一种良好的编程习惯。

软件工程师消除编译警告应当成为份内之事，而非可做可不做之事。清除自己代码的每一个警告既是负责的表现，也是专业做事的需要。

正因为不少缺陷是因为工程师忽视编译警告而造成的，所以，编译器的发展趋势是将更多的警告变成了错误。这也是为什么有些软件使用低版本的编译器能编译成功，而换成高版本的编译器后却总是编译失败的原因。

27.2　小结

编码质量对软件产品的质量同样具有决定性的作用，再好的软件设计也离不开高质量代码的表达。除了运用后面将要涉及的各种质量保证方法外，工程师的编程好习惯将有助于提高编码质量。另外，请不要忘记在第 11 章还指出了“永远将头文件作为（函数和变量的）定义和引用的桥梁”这一编程好习惯。



第 28 章 单元测试，被忽视的质量保证方法

相信没有真正体会到单元测试好处的读者一看到“单元测试”这几个字，可能会出现以下两种反应之一：

（1）由于没有单元测试的经验，因此对于采用这一方法去保证软件质量很好奇，也迫切地想了解这一方法在项目中如何实施。

（2）曾经使用过单元测试方法但效果并不好，因此一看到“单元测试”这几个字的反应是“没用”。

如果读者是第一种反应那很好，通过本章将掌握单元测试应如何实施；如果读者是第二种反应，那希望在阅读完本章及后续的章节后改变“单元测试没有用”的观念。

28.1 警惕单元测试无用论

造成“单元测试无用论”的第一个原因是，运用这一方法的时机不恰当。不少项目在一开始真正关心质量的人很少，更谈不上采用一整套的方法论去保证质量了。产品在开发出来后发现质量存在极大的问题，那时项目管理人员就想到了单元测试。于是，一声令下，整个项目开始做单元测试。为了对一个模块进行单元测试，首先要做的是把它分离出来，假如模块之间的耦合度很大的话，痛苦可想而知。

单元测试是一项耗时的工作，身处质量困境的管理者却希望在短期看到效果，或许单元测试还没有做到位管理层就等不及了，又想到换另一种方法——购买那些宣称能有效提高软件质量的测试工具。在前面的章节中我们谈到了一个比方，即将软件开发好了以后再做单元测试比作建好房子以后再检查用于建筑的砖是否是合格的，这个比方也很好地揭示了不恰当时机引入单元测试所带来的痛苦。正确的做法是，在项目的开始之初就引入单元测试。对于以前没有部署单元测试的项目，先只对新增加的、相对独立的模块做单元测试，并逐渐涵盖老代码。

第二个导致"单元测试无用论"的原因是，方法没有运用到位。要保证单元测试的有效性一定要引入另一个概念——代码覆盖（code coverage），在第 29 章将对它做专门介绍。只有将单元测试方法与代码覆盖结合在一起，才有可能保证单元测试的效果。

另外，"单元测试无用论"的存在也有一定的客观因素，那就是维护成本。单元测试一旦在项目中部署所带来的成本并非只发生在新增项目源代码时，而是只要对项目代码做了更改就得对相应的单元测试代码进行修订以适应变化。单元测试的部署将直接导致项目规模变大，这是不可小视的成本增加。那能否去除对单元测试代码的后期维护以减小因为引入单元测试所造成的后期成本呢？很遗憾，不行！因为一旦这样做，结局就是前功尽弃。

28.2　一个简单但不完善的单元测试例子

为了介绍单元测试是什么，先假设已经存在一个设计好的模块——双向链表模块。为了节省篇幅，图 28.1 只列出这个双向链表模块的头文件。

```
00029: #include "primitive.h"
00030:
00031: /* DLL stands for Double-Linked List */
00032:
00033: typedef struct dll_node {
00034:     struct dll_node *prev_;
00035:     struct dll_node *next_;
00036: } dll_node_t, *dll_node_handle_t;
00037:
00038: typedef struct {
00039:     dll_node_t *head_;
00040:     dll_node_t *tail_;
00041:     usize_t count_;
00042: } dll_t, *dll_handle_t;
00043:
00044: typedef bool (*traverse_callback_t)(dll_t *, dll_node_t *, void *);
00045:
00046: #ifdef __cplusplus
00047: extern "C" {
00048: #endif
00049:
00050: static inline void dll_init (dll_t *_p_dll)
00051: {
00052:     _p_dll->head_ = _p_dll->tail_ = 0;
00053:     _p_dll->count_ = 0;
00054: }
00055:
00056: static inline void dll_node_init (dll_node_t *_p_node)
00057: {
00058:     _p_node->next_ = _p_node->prev_ = 0;
00059: }
00060:
00061: static inline usize_t dll_size (const dll_t *_p_dll)
00062: {
00063:     return _p_dll->count_;
00064: }
```

```
00065:
00066: static inline dll_node_t *dll_head (const dll_t *_p_dll)
00067: {
00068:     return _p_dll->head_;
00069: }
00070:
00071: static inline dll_node_t *dll_tail (const dll_t *_p_dll)
00072: {
00073:     return _p_dll->tail_;
00074: }
00075:
00076: static inline dll_node_t *dll_next (const dll_t *_p_dll, const dll_node_t *_p_node)
00077: {
00078:     UNUSED (_p_dll);
00079:     return _p_node->next_;
00080: }
00081:
00082: static inline dll_node_t *dll_prev (const dll_t *_p_dll, const dll_node_t *_p_node)
00083: {
00084:     UNUSED (_p_dll);
00085:     return _p_node->prev_;
00086: }
00087:
00088: void dll_insert_before (dll_t *_p_dll,dll_node_t *_p_ref,dll_node_t *_p_inserted);
00089: void dll_insert_after (dll_t *_p_dll,dll_node_t *_p_ref,dll_node_t *_p_inserted);
00090: void dll_push_head (dll_t *_p_dll, dll_node_t *_p_node);
00091: void dll_push_tail (dll_t *_p_dll, dll_node_t *_p_node);
00092: dll_node_t *dll_pop_head (dll_t *_p_dll);
00093: dll_node_t *dll_pop_tail (dll_t *_p_dll);
00094: void dll_remove (dll_t *_p_dll, const dll_node_t *_p_node);
00095: dll_node_t *dll_traverse (dll_t *_p_dll, traverse_callback_t _cb, void *_p_arg);
00096: dll_node_t *dll_traverse_reversely (dll_t *_p_dll, traverse_callback_t _cb,
00097:     void *_p_arg);
00098: void dll_merge (dll_t *_p_to, dll_t *_p_from);
00099: void dll_split (dll_t *_p_orig, dll_t *_p_derived, dll_node_t *_p_breakpoint,
00100:     bool _breakpoint_belongs_to_orig);
00101:
00102: #ifdef __cplusplus
00103: }
00104: #endif
```

图 28.1

现在就以 dll_init()函数为例，来考虑如何为这个函数设计测试用例。dll_init()的实现是如此简单，以致于很容易让人觉得对它进行单元测试是多余的。但是，从单元测试的角度来看，只要不存在可行性问题就不应考虑因为简单而不对其进行验证。这从一致性的角度来说并不好，毕竟简单与复杂的评判会因人而易。再则，如果放弃对之进行验证，将降低代码覆盖率（参见第 29 章）。

做单元测试需要通过编写程序的方式来完成，所编写的用于测试被测模块的代码又称为单元测试用例，或简称为测试用例。图 28.2 是 dll_init()函数的测试用例。

```
00185: int main (int argc, char *argv[])
00186: {
00187:     dll_t list;
```

```
00188:
00189:     UNUSED (argc);
00190:     UNUSED (argv);
00191:
00192:     list.count_ = 0x5a5a;
00193:     list.head_ = (dll_node_t *)0xaaaa;
00194:     list.tail_ = (dll_node_t *)0xbbbb;
00195:
00196:     dll_init (&list);
00197:
00198:     // check the size of dll
00199:     if (list.count_ != 0) {
00200:         return -1;
00201:     }
00202:
00203:     // check the head of dll
00204:     if (list.head_ != 0) {
00205:         return -1;
00206:     }
00207:
00208:     // check the tail of dll
00209:     if (list.tail_ != 0) {
00210:         return -1;
00211:     }
00212:
00213:     return 0;
00214: }
```

图 28.2

图中在调用第 196 行的 dll_init()函数之前，故意将 list 变量中的各个成员变量初始化为一个非 0 值，如第 192～194 行所示。然后在对 list 变量调用完了 dll_init()函数以后，检查各成员是否被初始化成了 0，以判断 dll_init()函数是否真的起作用。注意，该测试程序里还有一个约定，返回-1 表示测试失败，返回 0 表示测试成功。

请注意，dll_init()函数测试用例的设计是基于我们了解 dll_init()函数的具体实现的，所以说单元测试是一种白盒测试。

通过 dll_init()函数的测试用例，相信读者能立即明白什么是单元测试。单元测试是通过编写程序来验证被测代码的行为是否如所设计的那样。图 28.2 的这个小程序需要与被测试程序(即 dll.c)一起编译到一个可执行文件中，并通过运行测试程序获得测试结果。后面，我们会通过使用 make 来简化这一操作。

对于这个小小的单元测试程序，存在如下有待改进的地方。

(1) 如果对于每一次检查都采用直接写 if 语句的形式将造成大量的冗余代码，且测试用例的编写效率也会很低。第 199～211 行的代码就很好地反映了这一问题。

(2) 直接通过观察程序是否返回 0 或者-1 来判断所有的测试是否通过或存在部分不通过并不直观。比如，如果图 28.2 的程序在运行时返回了-1，那到底是哪一步测试出了问题？毫无疑问，我们需要更直观的方式来展示哪一步成功或哪一步失败。

（3）如果一个测试程序存在 100 次判定，其中出现了 3 次失败，那最终显示一个成功百分比会比较直观，比如可以显示 97%的测试成功了。

所有这些有待提高的地方都表明，一个用于简化编写单元测试程序和提高单元测试结果可读性的单元测试框架很有必要。

28.3　构建单元测试框架

开源的单元测试框架有很多，比如 CppUnit、cxxtest 等。那我们为什么又要“重新造轮子”来构建自己的单元测试框架呢？为了简化！在作者看来，绝大部分的单元测试框架都复杂了点，以致于让人第一感觉有点望而生畏。这些框架有它被设计得复杂的原因，且初衷也是好的，比如提供“test suite”的管理等，但作者看来将单元测试程序与代码覆盖整合以后，这些概念的存在就显得不是很有意义了。与望而生畏相比，一个简单实用的单元测试框架往往能在项目团队中取得出乎意料的好结果，也更可能为团队创造出拥抱单元测试的文化。别忘了，简单、够用的工具是我们所追求的。

图 28.3 是一个简单但却有效的单元测试框架，它的实现主要是运用了 C 语言中的宏。对于每一个单元测试程序，要做的第一件事就是包含 unitest.h 头文件以便运用这一框架。接下来让我们看一看 unitest.h 头文件中都有些什么。

```
00029: #include <stdio.h>
00030:
00031: static int g_case_total;
00032: static int g_case_succeeded;
00033: static int g_case_failed;
00034:
00035: #define INTERNAL_USE_ONLY_SUCCEEDED(_a, _b) \
00036:     printf ("    Expected: \""#_a"\" == \""#_b"\"\n Result: Succeeded\n")
00037: #define INTERNAL_USE_ONLY_FAILED(_a, _b) \
00038:     printf ("    Expected: \""#_a"\" == \""#_b"\"\n (X) Result: Failed\n")
00039: #define INTERNAL_USE_ONLY_CASE_SHOW() \
00040:     printf ("Case %d =====>\n", g_case_total); \
00041:     printf ("    Location: %s:%d\n", __FILE__, __LINE__); \
00042:
00043: #define UNITEST_ERROR(_string) do { \
00044:     printf ("Error: "#_string"\n"); \
00045:     return -1;\
00046: } while (0)
00047:
00048: #define UNITEST_EQUALS(_a, _b) do { \
00049:     g_case_total ++; \
00050:     INTERNAL_USE_ONLY_CASE_SHOW (); \
00051:     if ((_a) == (_b)) { \
00052:         INTERNAL_USE_ONLY_SUCCEEDED (_a, _b); \
00053:         g_case_succeeded ++; \
00054:     } \
00055:     else { \
00056:         INTERNAL_USE_ONLY_FAILED (_a, _b); \
00057:         g_case_failed ++; \
```

```
00058:     } \
00059:     printf ("\n"); \
00060: } while (0)
00061:
00062: #define UNITEST_DIFFERS(_a, _b) do { \
00063:     g_case_total ++; \
00064:     INTERNAL_USE_ONLY_CASE_SHOW (); \
00065:     if ((_a) == (_b)) { \
00066:         INTERNAL_USE_ONLY_FAILED (_a, _b); \
00067:         g_case_failed ++; \
00068:     } \
00069:     else { \
00070:         INTERNAL_USE_ONLY_SUCCEEDED (_a, _b); \
00071:         g_case_succeeded ++; \
00072:     } \
00073:     printf ("\n"); \
00074: } while (0)
00075:
00076: #define UNITEST_LESS_THAN(_a, _b) do { \
00077:     g_case_total ++; \
00078:     INTERNAL_USE_ONLY_CASE_SHOW (); \
00079:     if ((_a) < (_b)) { \
00080:         INTERNAL_USE_ONLY_SUCCEEDED (_a, _b); \
00081:         g_case_succeeded ++; \
00082:     } \
00083:     else { \
00084:         INTERNAL_USE_ONLY_FAILED (_a, _b); \
00085:         g_case_failed ++; \
00086:     } \
00087:     printf ("\n"); \
00088: } while (0)
00089:
00090: #define UNITEST_LESS_THAN_EQUALS(_a, _b) do { \
00091:     g_case_total ++; \
00092:     INTERNAL_USE_ONLY_CASE_SHOW (); \
00093:     if ((_a) <= (_b)) { \
00094:         INTERNAL_USE_ONLY_SUCCEEDED (_a, _b); \
00095:         g_case_succeeded ++; \
00096:     } \
00097:     else { \
00098:         INTERNAL_USE_ONLY_FAILED (_a, _b); \
00099:         g_case_failed ++; \
00100:     } \
00101:     printf ("\n"); \
00102: } while (0)
00103:
00104: #define UNITEST_GREATER_THAN(_a, _b) do { \
00105:     g_case_total ++; \
00106:     INTERNAL_USE_ONLY_CASE_SHOW (); \
00107:     if ((_a) > (_b)) { \
00108:         INTERNAL_USE_ONLY_SUCCEEDED (_a, _b); \
00109:         g_case_succeeded ++; \
00110:     } \
00111:     else { \
00112:         INTERNAL_USE_ONLY_FAILED (_a, _b); \
00113:         g_case_failed ++; \
00114:     } \
00115:     printf ("\n"); \
00116: } while (0)
00117:
00118: #define UNITEST_GREATER_THAN_EQUALS(_a, _b) do { \
00119:     g_case_total ++; \
```

```
00120:    INTERNAL_USE_ONLY_CASE_SHOW (); \
00121:    if ((_a) >= (_b)) { \
00122:       INTERNAL_USE_ONLY_SUCCEEDED (_a, _b); \
00123:       g_case_succeeded ++; \
00124:    } \
00125:    else { \
00126:       INTERNAL_USE_ONLY_FAILED (_a, _b); \
00127:       g_case_failed ++; \
00128:    } \
00129:    printf ("\n"); \
00130: } while (0)
00131:
00132: void unitest_bar ()
00133: {
00134:    printf ("**************************************************\n");
00135:    printf ("* Unit Test                                      *\n");
00136:    printf ("**************************************************\n");
00137:    printf ("\n");
00138: }
00139:
00140: int unitest_report ()
00141: {
00142:    printf ("\n");
00143:    printf ("**************************************************\n");
00144:    printf ("* Test Report                                    *\n");
00145:    printf ("**************************************************\n");
00146:
00147:    if (0 == g_case_total) {
00148:       printf ("No test case is run\n");
00149:       goto pass;
00150:    }
00151:    printf ("   Total: %d\n", g_case_total);
00152:    printf ("Succeeded: %d (%d%%)\n", g_case_succeeded,
                 g_case_succeeded*100/g_case_total);
00153:    printf ("**************************************************\n");
00154:    printf ("\n");
00155:
00156:    if (g_case_total != g_case_succeeded) {
00157:       printf ("\n");
00158:       printf (":-( Failed!\n");
00159:       printf ("\n");
00160:       return -1;
00161:    }
00162: pass:
00163:    printf ("\n");
00164:    printf (":-) Passed\n");
00165:    printf ("\n");
00166:    return 0;
00167: }
00168:
00169: #ifndef HAVE_MAIN
00170: void unitest_main (int _argc, char *_argv[]);
00171:
00172: int main (int _argc, char *_argv[])
00173: {
00174:    unitest_bar ();
00175:    unitest_main (_argc, _argv);
00176:    return unitest_report ();
00177: }
00178: #endif
```

图 28.3

第 31～33 行定义了三个变量，分别用于记录总的测试用例数、成功的测试用例数和失败的测试用例数。统计这些信息的目的是，在测试程序运行的最后计算和显示成功比例。接下来是几个宏的定义（以“INTERNEL_USE_ONLY_”为前缀的宏只能在 unitest.h 内部使用），它们的功能分别是：

- INTERNEL_USE_ONLY_SUCCEEDED(_a,_b)：在终端上打印出一条测试成功的信息。_a 和_b 参数的具体所指可以通过后面的例子了解。
- INTERNEL_USE_ONLY_FAILED(_a,_b)：在终端上打印出一条测试失败的信息。
- INTERNEL_USE_ONLY_CASE_SHOW()：在终端上打印每一个测试用例的编号，以及它在测试源代码中的行号。当出现测试失败时，这一显示信息有助于找到出错的具体位置。
- UNITEST_ERROR(_string)：将_string 字符串显示在终端上后，通过返回-1 终止单元测试程序的运行。-1 的返回意味着测试失败了。
- UNITEST_EQUALS(_a,_b)：用于比较_a 和_b 两个值是否相等，相等表示成功。在这个宏的定义中，第 49 行更新总的测试用例数，也就是说，UNITEST_EQUALS()宏每被调用一次，就被当做增加了一个测试用例。第 51 行是 if 判断语句，用于检查_a 和_b 是否相等。由于 UNITEST_EQUALS()是一个宏，所以_a 和_b 是什么完全依赖于宏被调用时的上下文，它们可以是变量、常量乃至函数调用。如果_a 和_b 相等，则说明测试成功，因此在第 52 行打印出测试成功的信息，并在第 53 行对成功的用例数进行更新。类似地，当测试不成功时，即_a 和_b 并不相等时，需要打印出测试失败的信息（第 56 行），并对测试用例统计变量进行更新（第 57 行）。
- UNITEST_DIFFERS(_a,_b)：用于判定_a 和_b 是否不相等，不相等则表示成功。有了前面对 UNITEST_EQUALS()宏的介绍后，相信读者能很容易理解它的实现，后面不再累述。
- UNITEST_LESS_THAN(_a,_b)：用于判定_a 是否小于_b，当_a 小于_b 时表示成功。
- UNITEST_LESS_THAN_EQUALS(_a,_b)：用于判定_a 是否小于等于_b，当_a 小于等于_b 时表示成功。
- UNITEST_GREATER_THAN(_a,_b)：用于判定_a 是否大于_b，当_a 大于_b 时表示成功。
- UNITEST_GREATER_THAN_EQUALS(_a,_b)：用于判定_a 是否大于等于_b，当_a 大于等于_b 时表示成功。

除了这几个宏外，在 unitest.h 头文件中还定义了如下几个函数。

- unitest_bar()：用于打印单元测试报告的抬头。
- unitest_report()：用于打印单元测试的成功率。当存在失败的测试用例时，这个函数返回-1，否则返回 0。
- main()：每一个单元测试程序都是一个可执行文件，因此需要一个 main()入口函数。在 main()函数的实现中，显然 unitest_main()函数需要我们根据被测模块去实现，实现

unitest_main()函数的过程其实就是设计测试用例的过程。请注意，main()函数在 HAVE_MAIN 宏（第 169 行）的控制之下，如果一个单元测试源文件因为某种原因需要定义自己的 main()函数时，需要在包含 unitest.h 文件之前定义 HAVE_MAIN 宏，让 unitest.h 文件中定义的 main()函数“消失”。

下面看一看如何使用这个框架来编写单元测试程序，图 28.4 是运用这一框架重新编写的较完整的双向链表模块的单元测试程序。

```
00026: #include "unitest.h"
00027: #include "dll.h"
00028:
00029: static int KEY_HEAD = 3344;
00030: static int KEY_MIDDLE = 5566;
00031: static int KEY_TAIL = 7788;
00032:
00033: typedef struct {
00034:     dll_node_t node_;
00035:     int key_;
00036: } ut_node_t;
00037:
00038: static bool find_by_key (dll_t *_p_list, dll_node_t *_p_node, void *_p_arg)
00039: {
00040:     ut_node_t *p_test_node = (ut_node_t *)_p_node;
00041:     int *p_key = (int *)_p_arg;
00042:
00043:     UNUSED (_p_list);
00044:
00045:     if ((*p_key) == p_test_node->key_) {
00046:         return false;
00047:     }
00048:     return true;
00049: }
00050:
00051: void unitest_main (int argc, char *argv[])
00052: {
00053:     dll_t list, list_merged;
00054:     ut_node_t node_head, node_middle, node_tail;
00055:
00056:     UNUSED (argc);
00057:     UNUSED (argv);
00058:
00059:     // prepare the node structure
00060:     node_head.node_.next_ = (dll_node_t *)0xaaaa;
00061:     node_head.node_.prev_ = (dll_node_t *)0xbbbb;
00062:
00063:     node_head.key_ = KEY_HEAD;
00064:     node_middle.key_ = KEY_MIDDLE;
00065:     node_tail.key_ = KEY_TAIL;
00066:
00067:     // for testing dll_node_init ()
00068:     dll_node_init (&node_head.node_);
00069:     UNITEST_EQUALS (node_head.node_.next_, 0);
00070:     UNITEST_EQUALS (node_head.node_.prev_, 0);
```

```
00071:
00072:     list.count_ = 0x5a5a;
00073:     list.head_ = (dll_node_t *)0xaaaa;
00074:     list.tail_ = (dll_node_t *)0xbbbb;
00075:
00076:     // for testing dll_size ()
00077:     UNITEST_EQUALS (dll_size (&list), 0x5a5a);
00078:
00079:     // for testing dll_head ()
00080:     UNITEST_EQUALS (dll_head (&list), (dll_node_t *)0xaaaa);
00081:
00082:     // for testing dll_tail ()
00083:     UNITEST_EQUALS (dll_tail (&list), (dll_node_t *)0xbbbb);
00084:
00085:     // for testing dll_init ()
00086:     dll_init (&list);
00087:     UNITEST_EQUALS (dll_size (&list), 0);
00088:     UNITEST_EQUALS (dll_head (&list), 0);
00089:     UNITEST_EQUALS (dll_tail (&list), 0);
00090:
00091:     // for testing dll_push_head ()
00092:     dll_push_head (&list, &node_head.node_);
00093:     UNITEST_EQUALS (dll_size (&list), 1);
00094:     UNITEST_EQUALS (dll_head (&list), &node_head.node_);
00095:     UNITEST_EQUALS (dll_tail (&list), dll_head (&list));
00096:
00097:     // for testing dll_push_tail ()
00098:     dll_push_tail (&list, &node_tail.node_);
00099:     UNITEST_EQUALS (dll_size (&list), 2);
00100:     UNITEST_EQUALS (dll_head (&list), &node_head.node_);
00101:     UNITEST_EQUALS (dll_tail (&list), &node_tail.node_);
00102:
00103:     // for testing dll_next ()
00104:     UNITEST_EQUALS (dll_next (&list, &node_head.node_), &node_tail.node_);
00105:
00106:     // for testing dll_prev ()
00107:     UNITEST_EQUALS (dll_prev (&list, &node_tail.node_), &node_head.node_);
00108:
00109:     // for testing dll_insert_after ()
00110:     dll_insert_after (&list, &node_head.node_, &node_middle.node_);
00111:     UNITEST_EQUALS (dll_size (&list), 3);
00112:     UNITEST_EQUALS (dll_head (&list), &node_head.node_);
00113:     UNITEST_EQUALS (dll_tail (&list), &node_tail.node_);
00114:     UNITEST_EQUALS (dll_next (&list, &node_head.node_), &node_middle.node_);
00115:
00116:     // for testing dll_remove ()
00117:     dll_remove (&list, &node_middle.node_);
00118:     UNITEST_EQUALS (dll_size (&list), 2);
00119:     UNITEST_EQUALS (dll_head (&list), &node_head.node_);
00120:     UNITEST_EQUALS (dll_tail (&list), &node_tail.node_);
00121:     UNITEST_EQUALS (dll_next (&list, &node_head.node_), &node_tail.node_);
00122:
00123:     // for testing dll_insert_before ()
00124:     dll_insert_before (&list, &node_tail.node_, &node_middle.node_);
00125:     UNITEST_EQUALS (dll_size (&list), 3);
00126:     UNITEST_EQUALS (dll_head (&list), &node_head.node_);
00127:     UNITEST_EQUALS (dll_tail (&list), &node_tail.node_);
00128:     UNITEST_EQUALS (dll_next (&list, &node_head.node_), &node_middle.node_);
```

```
00129:
00130:     // for testing dll_pop_head ()
00131:     UNITEST_EQUALS (dll_pop_head (&list), &node_head.node_);
00132:     UNITEST_EQUALS (dll_size (&list), 2);
00133:     UNITEST_EQUALS (dll_head (&list), &node_middle.node_);
00134:
00135:     // for testing dll_pop_tail ()
00136:     UNITEST_EQUALS (dll_pop_tail (&list), &node_tail.node_);
00137:     UNITEST_EQUALS (dll_size (&list), 1);
00138:     UNITEST_EQUALS (dll_tail (&list), &node_middle.node_);
00139:
00140:     // for testing dll_traverse ()
00141:     UNITEST_EQUALS (dll_traverse (&list, find_by_key, (void *) &KEY_HEAD), 0);
00142:     UNITEST_EQUALS (dll_traverse (&list, find_by_key,
00143:         (void *) &KEY_MIDDLE), (dll_node_t *)&node_middle.node_);
00144:     UNITEST_EQUALS (dll_traverse (&list, find_by_key, (void *) &KEY_TAIL), 0);
00145:     UNITEST_EQUALS (dll_traverse (&list, (traverse_callback_t)0,
          (void *) &KEY_TAIL), 0);
00146:
00147:     // for testing dll_merge ()
00148:     dll_init (&list_merged);
00149:     dll_push_head (&list_merged, &node_head.node_);
00150:     dll_push_tail (&list_merged, &node_tail.node_);
00151:     dll_merge (&list, &list_merged);
00152:     UNITEST_EQUALS (dll_size (&list), 3);
00153:     UNITEST_EQUALS (dll_head (&list), &node_middle.node_);
00154:     UNITEST_EQUALS (dll_tail (&list), &node_tail.node_);
00155:     UNITEST_EQUALS (dll_next (&list, &node_middle.node_), &node_head.node_);
00156:     UNITEST_EQUALS (dll_size (&list_merged), 0);
00157:     UNITEST_EQUALS (dll_head (&list_merged), 0);
00158:     UNITEST_EQUALS (dll_tail (&list_merged), 0);
00159:     dll_merge (&list_merged, &list);
00160:     UNITEST_EQUALS (dll_size (&list_merged), 3);
00161:     UNITEST_EQUALS (dll_head (&list_merged), &node_middle.node_);
00162:     UNITEST_EQUALS (dll_tail (&list_merged), &node_tail.node_);
00163:     UNITEST_EQUALS (dll_next (&list_merged, &node_middle.node_), &node_head.node_);
00164:     //dll_merge (&list_merged, &list);
00165:
00166:     // for testing dll_split ()
00167:     dll_split (&list_merged, &list, &node_head.node_, false);
00168:     UNITEST_EQUALS (dll_size (&list_merged), 1);
00169:     UNITEST_EQUALS (dll_head (&list_merged), &node_middle.node_);
00170:     UNITEST_EQUALS (dll_tail (&list_merged), &node_middle.node_);
00171:     UNITEST_EQUALS (dll_size (&list), 2);
00172:     UNITEST_EQUALS (dll_head (&list), &node_head.node_);
00173:     UNITEST_EQUALS (dll_tail (&list), &node_tail.node_);
00174:     dll_merge (&list_merged, &list);
00175:     dll_split (&list_merged, &list, &node_head.node_, true);
00176:     UNITEST_EQUALS (dll_size (&list_merged), 2);
00177:     UNITEST_EQUALS (dll_head (&list_merged), &node_middle.node_);
00178:     UNITEST_EQUALS (dll_tail (&list_merged), &node_head.node_);
00179:     UNITEST_EQUALS (dll_size (&list), 1);
00180:     UNITEST_EQUALS (dll_head (&list), &node_tail.node_);
00181:     UNITEST_EQUALS (dll_tail (&list), &node_tail.node_);
00182: }
```

图 28.4

任何一个单元测试程序的代码一定要包含 unitest.h 头文件和被测模块的头文件（这里是 dll.h），并在其中实现 unitest_main()函数。unitest_main()函数的实现，就是采用 unitest.h 头文件中定义的宏对相应的数据结构、变量或函数返回值进行判定。对于图中 unitest_main()函数的实现，相信读者通过阅读能很快地领会。

请注意图中的第 87～89 行，它们对应于上一节所讨论的简单测试程序（图 28.2）的第 199～211 行。很明显，在运用了单元测试框架以后，所编写的测试代码更加简练了。

总体说来，单元测试程序中的一个测试用例无非就包含下面三个步骤。

（1）构造测试条件。

（2）调用被测函数。

（3）检查测试结果。

编写单元测试用例，有点让人感觉是在“拼拼凑凑”。这是因为单元测试很重要的一个目的就是检查被测模块的边边角角，而这不可避免地需要拼凑验证条件。

28.4　无缝整合单元测试

在 26.3 节指出，软件质量保证需要系统性的方法论，而打造系统性的方法论需要将工具和流程进行无缝整合。我们从事开发工作是基于开发环境的，因此无缝整合也应当是基于开发环境的。

在第 3 章介绍 Makefile 时，创建了 simple、complicated 和 huge 三个项目以辅助 make 的知识介绍。从本章开始，我们将沿用 huge 项目中的知识点，在探讨质量保证方法的同时，继续深入运用 make 实现将工具和流程无缝整合到开发环境中。

尽管 huge 项目中所创建的 Makefile 已经很实用了，但它离真实的项目开发环境还存在一定的距离。比如，在真实的开发环境中，可能需要编译 release 版本软件进行官方发布，或者需要编译 debug 版本软件进行开发调试。另外，真实的项目中编译系统可能需要同时支持 C 文件和 C++文件的编译，而对于嵌入式软件开发它还得支持汇编程序的编译。

由此看来，需要对 huge 项目的编译系统进行改造，改造过程并不打算细讲，改造的结果读者可以从 embedded/buildv1 目录中找到。只要读者完整地阅读并理解了第 3 章的内容，那么就已经完全具备了理解 embedded/buildv1 目录下 Makefile 的能力。从本章开始，所有 Makefile 的更改都是基于 embedded/buildv1 目录中这一版本的。

尽管如何获得 embedded/buildv1 目录下编译系统的细节不打算介绍，但对 huge 项目与之的区别进行扼要的概括还是很有必要的，因为这有助于梳理新编译系统的脉络。新编译系统与 huge 项目的存在以下差别。

（1）huge 项目中的编译规则只支持 C 程序的编译，且规则是存放在 make.rule 文件中的。但新编译系统中，编译规则除了支持 C 程序外，还支持 C++和汇编程序。C 程序和 C++程序的编译规则分别存放在 c.rule 和 c++.rule 两个文件中，在 c.rule 文件中又包含了汇编程序的编译规则。c.rule 和 c++.rule 需要根据被编译的源程序是用 C 还是 C++语言进行正确包含。图 28.5 分别示例说明了 embedded 项目中两个不同的 Makefile，上面一个是针对 C++程序的（注意其中的第 13 行），而下面的是针对 C 语言的（注意其中的第 8 行）。

```
[illegible]/platform/device/src/Makefile
00001: EXE =
00002: LIB = libdevice.a
00003:
00004: INCLUDE_DIRS = \
00005:         $(ROOT)/code/platform/common/inc \
00006:         $(ROOT)/code/platform/arch/x86/simulator/inc \
00007:         $(ROOT)/code/platform/sync/v3/inc \
00008:         $(ROOT)/code/platform/task/v3/inc \
00009:         $(ROOT)/code/platform/device/inc \
00010:
00011: LINK_LIBS =
00012:
00013: include $(BUILD)/c.rule
00014:
```

```
[illegible]/err2str/Makefile
00001: EXE = err2str.exe
00002: LIB =
00003:
00004: INCLUDE_DIRS =
00005:
00006: LINK_LIBS =
00007:
00008: include $(BUILD)/c++.rule
00009:
```

图 28.5

（2）新编译系统支持 release 和 debug 两个软件版本的编译。两个软件版本的区别相信读者很清楚，这与软件是否进行优化有关。由于 release 版本软件采用了编译器的优化技术，因此它的运行速度更快。也因为这一原因，使得它不适合进行软件开发调试，因为优化技术为了获得软件的执行效率会打破 C/C++语言与汇编代码的直接映射。

（3）在新编译系统中，当编译 release 版本软件时，所有的目标文件和依赖关系文件将放入源程序所在目录的 robjs 子目录中，生成的最终库文件和可执行文件则放在 embedded/buildv1/ release 目录中；当编译 debug 版本软件时，目标文件和依赖关系文件将放入源程序所在目录的 dobjs 子目录中，embedded/ buildv1/debug 目录则是最终库文件和可执行文件的存放位置。

由于后面的连续几章都涉及对 Makefile 的修改，对于各章的 Makefile 将放入不同的目录。比如，本章的 Makefile 将放入 embedded/buildv2 目录中，高版本目录下的 Makefile 将包含低版

本中的内容。读者可以分别进入不同的目录直接对整个项目进行编译。

回到图 28.4 中的 unitest_dll.c 文件，下面看如何通过更改 Makefile 以最终编译出 unitest_dll.exe 可执行文件。对于修改后的 Makefile，我们希望通过执行“make unitest”命令就能编译出所有的单元测试可执行文件。

在更改 embedded/buildv1 目录下的 Makefile 以获得 embedded/buildv2 目录下的之前，需要定义一些规则来简化 Makefile 的设计。

28.4.1　维护规则

本书将使用如下几个单元测试方面的维护规则。实际上，每一个项目可以根据自己的喜好定义与这里不同的规则，但需要注意，规则的定义需要通过 Makefile 进行匹配。

（1）每一个被测文件所对应的单元测试源代码的文件名，都是在其前加上“unitest_”前缀。比如，dll.c 的单元测试代码的文件名应为 unitest_dll.c，dllht.c 的单元测试代码的文件名为 unitest_dllht.c。

（2）每一个单元测试代码文件都将编译出一个独立的可执行文件，且可执行文件名与单元测试程序的文件名前缀相同。比如，unitest_dll.c 将编译出 unitest_dll.exe 可执行文件，而 unitest_dllht.c 将编译出 unitest_dllht.exe 可执行文件。

（3）单元测试代码与被测模块放在不同的目录中，且永远不会混放在同一目录中。

（4）所有被测模块都将以库的形式提供。在这种情形下，编译出一个单元测试可执行程序只需要两个输入文件，其中一个是单元测试用例文件，另一个则是包含被测文件编译而成的库。

（5）每一个单元测试可执行文件如果存在测试失败的情形，则返回-1，否则返回 0。这一规则其实已经隐含在前面介绍的单元测试框架中了。回忆一下，unitest_report()函数在发现存在测试用例失败的情形时，将返回-1，这个值也是整个 main()函数的返回值。

28.4.2　目录规划

单元测试源程序所存放的目录也需要规划，大致存在三种方式。第一种方式是，并不为单元测试代码文件提供独立的目录，而是将其与被测代码混放在一起，然后通过 Makefile 来指定哪些文件是单元测试代码，哪些文件是非测试代码。对于这种方式，如果项目不大可以接受，但当项目大了以后就会显得混乱，难以维护。

第二种方式如图 28.6 所示，即在模块的 inc 和 src 目录内增加一个与之平行的 unitest 目录，然后将相应模块的单元测试源文件放入其中。

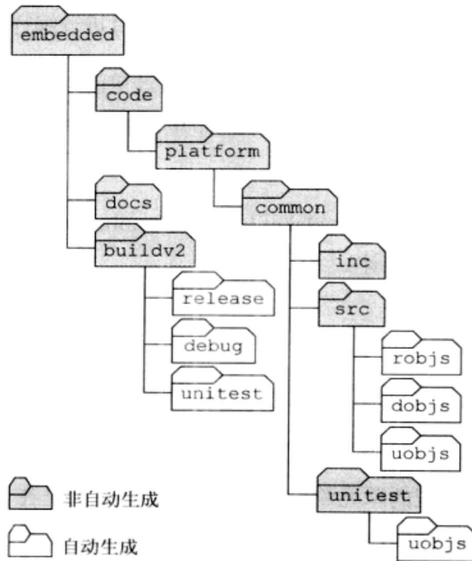


图 28.6

第三种方式则是在与 code 平行的目录级别上建立一个 tests 目录，然后在 tests 目录中采用与 code 目录下相同的目录结构来组织单元测试程序，如图 28.7 所示。

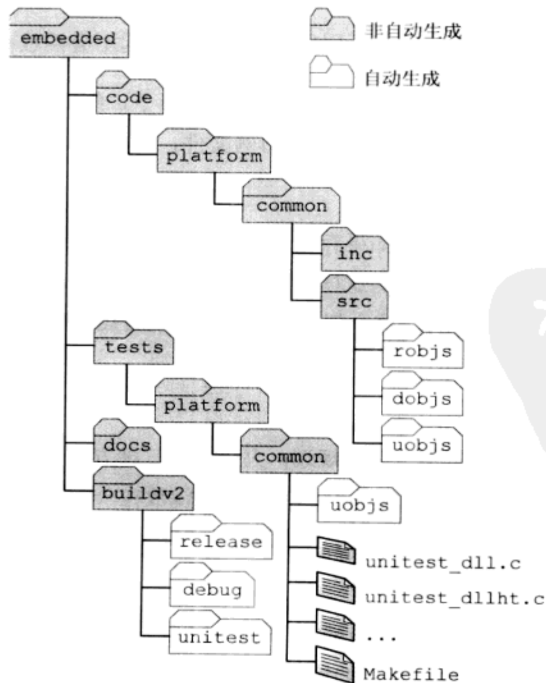


图 28.7

后两种方式并没有本质的区别，只是作者觉得最后一种方式更具集中管理的味道，因此在本书中也采用了它。需要注意，在 code 目录下存在的模块需要在 tests 下为之建立同名的目录，只不过单元测试程序通常是由各个独立的.c 文件组成的，因此不需要考虑建立 inc 和 src 两个目录。显然，在每一个存放单元测试源程序的目录中，还得放置一个 Makefile 文件。

图 28.7 还传递了其他的信息。例如，单元测试源程序所编译出来的目标文件，将被放入源程序所在的 uobjs 子目录中；最终生成的库和单元测试可执行程序，将被放入 embedded/buildv2 目录下的 unitest 子目录中。

将所有的单元测试可执行文件放入同一个目录中，可以方便我们运行各个单元测试程序，也方便编写 Makefile 实现单元测试自动化，后面读者将看到这些优点。

28.4.3 更改 Makefile

对 Makefile 进行修改所实现的功能分为两部分：一部分是通过运行“make unitest”命令生成单元测试可执行文件；另一部分是通过运行“make test”进行单元测试。让我们先从实现第一部分功能开始。

为了将单元测试整合到项目中，需要修改 embedded/buildv1 目录下的 c.rule、c++.rule 和 Makefile 三个文件，并将修改后的文件放入 embedded/buildv2 目录中。

由于 c.rule 和 c++.rule 的更改是完全雷同的，因此我们只需关注 c.rule 即可。图 28.8 是更改后的 c.rule 文件，图 28.9 则采用 UML 形式示例说明了所增加或修改的规则在依赖关系中的位置，以便读者能明白其具体作用。

```
00001: MKDIR = mkdir
00002: RM = rm
00003: AR = ar
00004: CC = gcc
00005:
00006: RMFLAGS = -fr
00007: ARFLAGS = crs
00008: CFLAGS += -Wall -std=gnu99
00009:
00010: ROOT := $(abspath $(ROOT))
00011: BUILD := $(abspath $(BUILD))
00012:
00013: ifeq ("$(MAKECMDGOALS)", "")
00014: MAKECMDGOALS = release
00015: endif
00016:
00017: ifeq ("$(MAKECMDGOALS)", "release")
00018: DIR_OBJS = robjs
00019: AR = ar
00020: CC = gcc
00021: CFLAGS += -gdwarf-2 -g3 -O2
00022: endif
00023:
```

```
00024: ifeq ("$(MAKECMDGOALS)", "debug")
00025: DIR_OBJS = dobjs
00026: AR = ar
00027: CC = gcc
00028: CFLAGS += -gdwarf-2 -g3
00029: endif
00030:
00031: ifeq ("$(MAKECMDGOALS)", "unitest")
00032: DIR_OBJS = uobjs
00033: AR = ar
00034: CC = gcc
00035: CFLAGS += -gdwarf-2 -g3 -DUNITEST
00036: endif
00037:
00038: DIR_TARGET = $(BUILD)/$(MAKECMDGOALS)
00039: DIRS = $(DIR_OBJS) $(DIR_TARGET)
00040:
00041: SRCS = $(wildcard *.c)
00042: ASMS = $(wildcard *.S)
00043: UTS = $(wildcard unitest_*.c)
00044: OBJS := $(addprefix $(DIR_OBJS)/, $(SRCS:.c=.o))
         $(addprefix $(DIR_OBJS)/, $(ASMS:.S=.o))
00045: DEPS := $(addprefix $(DIR_OBJS)/, $(SRCS:.c=.dep))
        $(addprefix $(DIR_OBJS)/, $(ASMS:.S=.dep))
00046: RMS = robjs dobjs uobjs
00047:
00048: ifneq ("$(EXE)", "")
00049: EXE := $(addprefix $(DIR_TARGET)/, $(EXE))
00050: RMS += $(EXE)
00051: endif
00052: ifneq ("$(UTS)", "")
00053: UTS := $(addprefix $(DIR_TARGET)/, $(UTS:.c=.exe))
00054: RMS += $(UTS)
00055: endif
00056: ifneq ("$(LIB)", "")
00057: LIB := $(addprefix $(DIR_TARGET)/, $(LIB))
00058: RMS += $(LIB)
00059: endif
00060:
00061: ifeq ("$(wildcard $(DIR_OBJS))", "")
00062: DEP_DIR_OBJS := $(DIR_OBJS)
00063: endif
00064: ifeq ("$(wildcard $(DIR_TARGET))", "")
00065: DEP_DIR_TARGET := $(DIR_TARGET)
00066: endif
00067:
00068: ifneq ("$(INCLUDE_DIRS)", "")
00069: INCLUDE_DIRS := $(addprefix -I, $(INCLUDE_DIRS))
00070: INCLUDE_DIRS := $(strip $(INCLUDE_DIRS))
00071: endif
00072:
00073: ifneq ("$(LINK_LIBS)", "")
00074: LINK_LIBS := $(strip $(LINK_LIBS))
00075: LIB_ALL := $(notdir $(wildcard $(DIR_TARGET)/*))
00076: LIB_FILTERED := $(addsuffix %, $(addprefix lib, $(LINK_LIBS)))
00077: $(eval DEP_LIBS = $(filter $(LIB_FILTERED), $(LIB_ALL)))
00078: DEP_LIBS := $(addprefix $(DIR_TARGET)/, $(DEP_LIBS))
00079: LINK_LIBS := $(addprefix -l, $(LINK_LIBS))
00080: endif
00081:
```

```
00082: .PHONY: release debug clean unitest
00083: release debug unitest: $(EXE) $(UTS) $(LIB)
00084: clean:
00085:     -$(RM) $(RMFLAGS) $(RMS)
00086:
00087: ifneq ("$(MAKECMDGOALS)", "clean")
00088: include $(DEPS)
00089: endif
00090:
00091: $(DIRS):
00092:     $(MKDIR) $@
00093: $(EXE): $(DEP_DIR_TARGET) $(OBJS) $(DEP_LIBS)
00094:     $(CC) -L$(DIR_TARGET) -o $@ $(filter %.o, $^) $(LINK_LIBS)
00095: $(UTS): $(DEP_DIR_TARGET) $(DEP_LIBS)
00096:     $(CC) -L$(DIR_TARGET) -o $@ $(filter %.o, $^) $(LINK_LIBS)
00097: $(LIB): $(DEP_DIR_TARGET) $(OBJS)
00098:     $(AR) $(ARFLAGS) $@ $(filter %.o, $^)
00099:
00100: $(DIR_OBJS)/%.o: $(DEP_DIR_OBJS) %.c
00101:     $(CC) $(CFLAGS) $(INCLUDE_DIRS) -o $@ -c $(filter %.c, $^)
00102: $(DIR_OBJS)/%.o: $(DEP_DIR_OBJS) %.S
00103:     $(CC) $(CFLAGS) $(INCLUDE_DIRS) -o $@ -c $(filter %.S, $^)
00104: $(DIR_OBJS)/%.dep: $(DEP_DIR_OBJS) %.c
00105:     @echo "Creating $@ ..."
00106:     @set -e ; \
00107:     $(RM) $(RMFLAGS) $@.tmp ; \
00108:     $(CC) $(INCLUDE_DIRS) -E -MM $(filter %.c, $^) > $@.tmp ; \
00109:     sed 's,\(.*\)\.o[ :]*,$(DIR_OBJS)/\1.o $@: ,g' < $@.tmp > $@ ;\
00110:     $(RM) $(RMFLAGS) $@.tmp ; \
00111:     if [ -n "$(UTS)" ]; then echo "$(DIR_TARGET)/$*.exe: $(DIR_OBJS)/$*.o" >>$@;fi
00112: $(DIR_OBJS)/%.dep: $(DEP_DIR_OBJS) %.S
00113:     @echo "Creating $@ ..."
00114:     @set -e ; \
00115:     $(RM) $(RMFLAGS) $@.tmp ; \
00116:     $(CC) $(INCLUDE_DIRS) -E -MM $(filter %.S, $^) > $@.tmp ; \
00117:     sed 's,\(.*\)\.o[ :]*,$(DIR_OBJS)/\1.o $@: ,g' < $@.tmp > $@ ;\
00118:     $(RM) $(RMFLAGS) $@.tmp ;
```

图 28.8

对 c.rule 的更改包含如下几处。

（1）第 3～4 行被移到或复制到了第 19～20 和 26～27 行，之所以进行这样的变化完全是因为新增的第 31～36 行。对于真实的嵌入式系统，当编译 release 和 debug 版本的程序时使用的是交叉编译器进行编译，而当编译 unitest 版本的程序时则需要使用非交叉编译器。除了编译器，用于生成库的工具 ar 也需要进行区分。由于本书所写的程序目前都是在主机上运行的，所以对于 release、debug 和 unitest 三个目标的 CC 和 AR 变量的值都相同。注意，在第 35 行的 CFLAGS 变量内，通过 gcc 的-D 选项定义了一个名为 UNITEST 的宏，这个宏的作用后面会看到。

（2）新增的第 43 行定义了一个 UTS 变量，这个变量的值是通过 wildcard 函数获得目录中的所有单元测试源文件的列表。从这里可以看出，前面定义的第一条维护规则在这里发挥了作用。

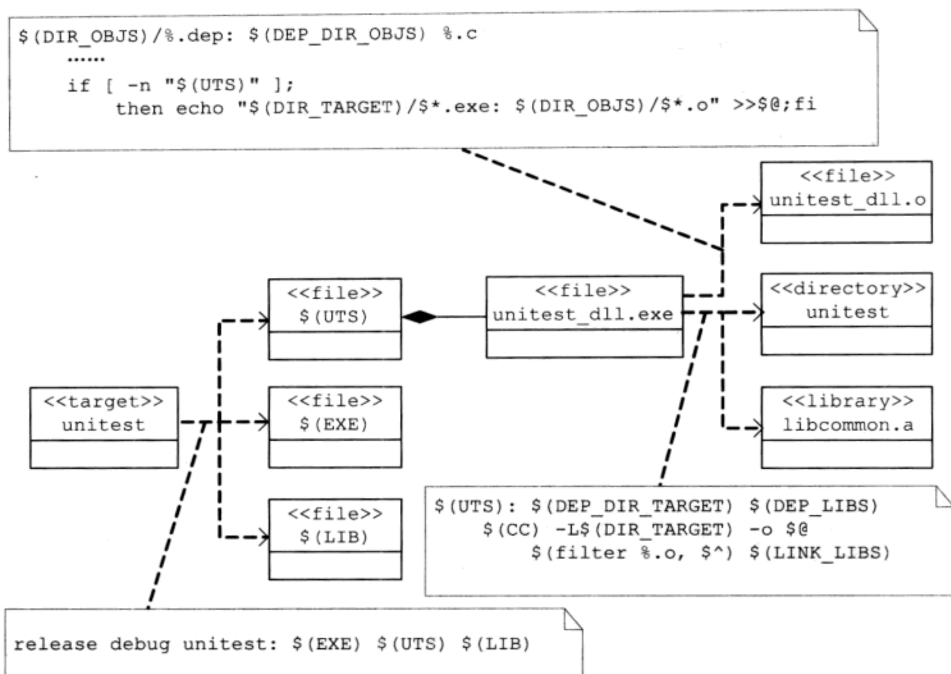


图 28.9

（3）新增的第 52～55 行用于判断 UTS 变量的值是否为空，如果不为空则在 UTS 变量内各子串前加上前缀，使各文件的路径变为绝对路径。要做到这一点，只需将 DIR_TARGET 变量的值作为前缀即可。在增加前缀时需要将.c 后缀变为.exe 后缀，也就是说，UTS 变量中保存的是所有要生成的单元测试可执行文件。

（4）修改的第 82～83 行增加了 unitest 假目标，以及让 release、debug 和 unitest 三个目标依赖于$(UTS)。其实，在构建 release 和 debug 目标时，它们不需要依赖$(UTS)，只有构建 unitest 目标时才需要依赖$(UTS)。将它们都放在一个规则中并不会出现问题，后面在讲解 embedded/buildv2/Makefile 时读者将明白为什么。写在同一个规则中的目的是为了简洁。

（5）新增的第 95 和 96 行定义了编译每一个单元测试可执行程序的规则。比如，将 unitest_dll.c 最终编译生成 unitest_dll.exe 就需要使用到这一规则。如果读者注意比较的话将发现，新增的两行与第 93 和 94 行很相似，只是少了对$(OBJS)的依赖。在 c.rule 中，$(OBJS)是当前目录下的所有.c 文件所对应的目标文件。如果目录下只存在 unitest_dll.c 和 unitest_dllht.c 两个文件，则$(OBJS)中将包含 unitest_dll.o 和 unitest_dllht.o 两个文件名。显然，在这种情形下，我们不希望 unitest_dll.exe 和 unitest_dllht.exe 两个单元测试程序都依赖于这两个目标文件，而应是 unitest_dll.exe 只依赖于 unitest_dll.o，以及 unitest_dllht.exe 只依赖于 unitest_dllht.o。这就是为什么新增规则中不让$(UTS)依赖于$(OBJS)的原因。

（6）新增第 111 行的作用是判断如果所生成的依赖关系文件是某单元测试源文件的，那么多生成一条用于描述单元测试可执行文件与目标程序之间的依赖关系。这一依赖关系是针对每一个单元测试源程序的。根据前面的第三条维护规则，UTS 变量只有当目录中存在且只存在单元测试源文件时才不为空。图 28.10 显示了通过使用 cat 命令所查看到的增加了这一改动后 unitest_dll.dep 文件的内容，其中的最后一行体现了更改的效果。

```
                    /embedded/buildv2
$ cat ../tests/platform/common/uobjs/unitest_dll.dep
uobjs/unitest_dll.o uobjs/unitest_dll.dep: unitest_dll.c \
    /embedded/code/platform/common/inc/unitest.h \
    /embedded/code/platform/common/inc/dll.h \
    /embedded/code/platform/common/inc/primitive.h
/embedded/buildv2/unitest/unitest_dll.exe: uobjs/unitest_dll.o
```

图 28.10

c.rule 文件有了这些更改以后，就支持单元测试可执行程序的编译了。下面还需对 Makefile 文件进行修改，修改后的内容如图 28.11 所示。

```
00001: ROOT = $(realpath ..)
00002: BUILD = $(realpath .)
00003:
00004: COMMON_DIRS = \
00005:     $(ROOT)/code/tools/err2str \
00006:     $(ROOT)/code/platform/common/src \
00007:     ......
00026:
00027: NONUT_DIRS = \
00028:     $(ROOT)/code/application/stackframe \
00029:     $(ROOT)/code/application/return \
00030:     ......
00049:
00050: UT_DIRS = \
00051:     $(ROOT)/tests/platform/common \
00052:     $(ROOT)/tests/platform/task \
00053:
00054: MAKE_DIRS := $(COMMON_DIRS)
00055:
00056: ifeq ("$(MAKECMDGOALS)", "")
00057: MAKECMDGOALS = release
00058: endif
00059:
00060: ifeq ("$(MAKECMDGOALS)", "release")
00061: MAKE_DIRS += $(NONUT_DIRS)
00062: endif
00063: ifeq ("$(MAKECMDGOALS)", "debug")
00064: MAKE_DIRS += $(NONUT_DIRS)
00065: endif
00066: ifeq ("$(MAKECMDGOALS)", "unitest")
00067: MAKE_DIRS += $(UT_DIRS)
00068: endif
00069: ifeq ("$(MAKECMDGOALS)", "clean")
00070: MAKE_DIRS += $(NONUT_DIRS) $(UT_DIRS)
00071: endif
```

```
00072:
00073: RM = rm
00074: RMFLAGS = -fr
00075: RMS = release debug unitest
00076:
00077: DIR_UNITEST = unitest
00078:
00079: .PHONY: release debug clean touch unitest test force
00080: release debug clean unitest:
00081:     @set -e; \
00082:     for DIR in $(MAKE_DIRS); \
00083:     do \
00084:         cd $$DIR && $(MAKE) -r ROOT=$(ROOT) BUILD=$(BUILD) $@;\
00085:     done
00086:     @set -e; \
00087:     if [ "$(MAKECMDGOALS)" = "clean" ] ; then $(RM) $(RMFLAGS) $(RMS) ; fi; \
00088:     if [ "$(MAKECMDGOALS)" = "unitest" ] ; then touch $(DIR_UNITEST)/unitested; fi
00089:     @echo ""
00090:     @echo ":-) Completed"
00091:     @echo ""
00092:
00093: UTS = $(wildcard $(DIR_UNITEST)/unitest_*.exe)
00094:
00095: ifeq ("$(MAKECMDGOALS)", "test")
00096: ifeq ("$(wildcard $(DIR_UNITEST)/unitested)", "")
00097: $(error Did you forget to run 'make unitest'?)
00098: endif
00099: endif
00100:
00101: $(DIR_UNITEST)/unitest_%: force
00102:     ./$@
00103: test: $(UTS)
00104:     @touch $(DIR_UNITEST)/tested
00105:
00106: force:
00107:
00108: touch:
00109:     @echo "Processing ..."
00110:     @find $(ROOT) -exec touch {} \;
00111:     @echo ""
00112:     @echo ":-) Done"
00113:     @echo ""
```

图 28.11

Makefile 文件中的变更包括：

（1）第 4～52 行的主要变化是，增加了 COMMON_DIRS、NONUT_DIRS 和 UT_DIRS 三个变量，分别用于存放需要构建的目录。注意，这三个变量的定义是站在单元测试这个角度的，与是否构建 release 或 debug 目标无关。COMMON_DIRS 变量中存放的是构建 unitest 或非 unitest 目标都需要编译的目录，NONUT_DIRS 存放的是只在构建非 unitest 目标时才需要编译的目录，UT_DIRS 则存放只有在构建 unitest 目标时才需要编译的目录。

（2）第 54～71 行则根据不同的构建目标，设置 MAKE_DIRS 变量，该变量中存放的目录路径是每一次构建真正要编译的。

（3）第 75 行则将 unitest 目录增加到 RMS 变量中，即在运行“make clean”命令时，也将 embedded/buildv2/unitest 目录删除。

（4）第 77 行的 DIR_UNITEST 变量用于指定 buildv2 目录下用于存放生成的可执行文件和库文件的目录名。在这里其值被设置为“unitest”。

（5）第 79 行则将 unitest、test 和 force 三个目标定义为假目标。unitest 目标用于构建单元测试可执行程序；而 test 目标用于执行每一个单元测试可执行程序，或者说它被用于做单元测试；force 目标所在的规则是一个空规则，它的作用后面会涉及。

（6）第 80 行在已有规则中增加 unitest 目标，且对规则增加第 88 行的内容。即当构建 unitest 目标时，在目标构建完成以后需要在 embedded/buildv2/unitest 目录下生成一个 unitested 文件，以表示成功运行过“make unitest”，后面会看到为什么要做这么一个动作。

（7）新增的第 93 行通过使用 wildcard 函数获取 embedded/ buildv2/unitest 目录下的所有单元测试可执行程序，以便后面一个接一个地运行它们，这里同样用到了前面定义的第二条维护规则。

（8）第 95～99 行的功能是，在运行“make test”进行单元测试时需要先检查之前是否已运行过“make unitest”。这是因为单元测试可执行程序只有在运行过“make unitest”以后才能生成，也只有这样才能做单元测试。前面提到的 unitesed 文件就在这里派上了用场。如果这个文件不存在，则说明“make unitest”没有被执行过，则在终端上打印出“Did you forget to run ‘make unitest’？”提示用户。这一设计完全是从可使用性的角度出发的。

（9）新增的第 101～102 行定义了执行每一个单元测试可执行程序的规则。当运行“make test”时，这一规则将会被最终运用，以逐个执行位于 buildv2/unitest 目录下的单元测试可执行程序。在运行各单元测试可执行程序的过程中，如果出现可执行程序返回失败（即可执行程序返回非 0 值）的情形时，make 将会自动终止，这与我们编译程序时如果出现编译错误则 make 自行终止是一样的，这是通过前面的第五条维护规则做到的。

（10）新增的第 101～104 行定义了一个新的目标——test，且让它依赖于$(UTS)。前面提到了，$(UTS)是指 embedded/buildv2/unitest 目录下所有单元测试可执行程序文件的例表。

（11）新增的第 106 行定义了一个 force 目标，在 Makefile 中，如果一个目标没有任何依赖，则在每一次 make 的过程中，所有依赖于它的目标每次无论如何都会被重新构建。由于各单元测试可执行程序都依赖于 force 目标（第 101 行），所以每次运行“make test”时，一定会执行每一个单元测试可执行程序。

图 28.12 示例说明了 Makefile 中新增规则在依赖关系中的作用。其中只以 unitest_dll.exe 为例，在实际情形中，$(UTS)内还可以包含其他的单元测试可执行程序。

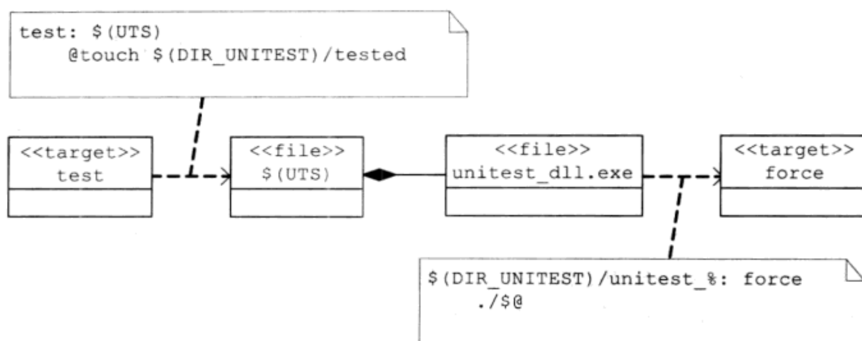


图 28.12

最后，别忘了在（只）存放有单元测试源文件的目录中创建一个 Makefile。比如，图 28.13 示例说明了 unitest_dll.c 文件所在 tests/platform/common 目录中 Makefile 的内容。

```
00001: EXE =
00002: LIB =
00003:
00004: INCLUDE_DIRS = $(ROOT)/code/platform/common/inc
00005:
00006: LINK_LIBS = common
00007:
00008: include $(BUILD)/c.rule
```

图 28.13

现在以 unitest_dll.c 文件为例，来说明前面的第四条维护规则。在该维护规则中指出，一个单元测试可执行程序的生成只需两个输入文件。一个是将编译好的被测文件包含在内的库，库的指定是通过图 28.8 中第 96 行的 LINK_LIBS 变量指定的。在图 28.13 中，LINK_LIBS 被设置为“common”，即 libcommon.a 库，unitest_dll.c 文件对应的 dll.c 的目标文件正是被包含在该库中的。另一个则是单元测试文件本身，相应的依赖关系是通过图 28.8 中第 111 行的命令生成的，即图 28.10 最后一行指定的依赖文件 uobjs/unitest_dll.o。

所有用于编译单元测试可执行程序的 Makefile 都像图 28.13 那样简单，每个 Makefile 只需正确设置 INCLUDE_DIRS 和 LINK_LIBS 两个变量即可。另外，当需要在目录中增加一个单元测试源程序文件时，并不需要对 Makefile 做任何的更新。

28.4.4 检查整合效果

接下来，让我们看一看整合后的效果。先运行“make unitest”，接着使用 ls 命令检查 unitest 目录下的文件，结果如图 28.14 所示。

```
$ make unitest
...... 显示结果有删减 ......
```

```
$ ls unitest
err2str.exe*        libcommon.a             libdevice.a
libmemoryv2.a       libsyncv2.a             libtaskv2.a
libtimerv2.a        unitest_bitmap.exe*     unitest_dllht.exe*
lib39vf040.a        libcpu.a                libflash.a
libmemoryv3.a       libsyncv3.a             libtaskv3.a
libtimerv3.a        unitest_clib.exe*       unitest_task.exe*
libboard.a          libctrlc.a              libmemoryv1.a
libsyncv1.a         libtaskv1.a             libtimerv1.a
libtimerv4.a        unitest_dll.exe*        unitested
```

图 28.14

图中有几个可执行文件是以“unitest_”为前缀的，这些就是编译生成的单元测试可执行程序。各个单元测试可执行程序既可以通过手动的方式逐一执行，也可以通过运行“make test”让 make 程序一个一个地自动运行。图 28.15 是所有测试都成功时，运行“make test”获得的结果。

```
                    /embedded/buildv2
$ make test
...... 显示结果有删减 ......
Case 60 ------>
    Location: unitest_dll.c:180
    Expected: "dll_head (&list)" == "&node_tail.node_"
      Result: Succeeded

Case 61 ------>
    Location: unitest_dll.c:181
    Expected: "dll_tail (&list)" == "&node_tail.node_"
      Result: Succeeded


**************************************************************
*                        Test Report                         *
**************************************************************
    Total: 61
Succeeded: 61 (100%)
**************************************************************


:-) Passed

...... 显示结果有删减 ......
Case 30 ------>
    Location: unitest_task.c:134
    Expected: "module_device (STATE_DESTROYING)" == "0"
      Result: Succeeded

Case 31 ------>
    Location: unitest_task.c:135
    Expected: "module_interrupt (STATE_DESTROYING)" == "0"
      Result: Succeeded


**************************************************************
*                        Test Report                         *
**************************************************************
    Total: 31
Succeeded: 31 (100%)
**************************************************************


:-) Passed
```

图 28.15

图 28.15 是本书第一次涉及单元测试框架的输出结果，读者可以对照测试代码和 unitest.h 了解每行语句的作用。从图中我们可以得到以下信息：

（1）对于每一个测试用例，在终端上可以看出其运行结果，内容包括用例在测试源程序中的行号、期望结果和最终运行结果。如果失败，则会在结果之前打上一个“X”，以使其醒目，在图 28.16 中可以找到一个用例测试失败的例子。

（2）每个单元测试可执行程序运行到最后时，会打印出一个汇总报告。报告内容包括一共有多少个用例、成功用例数及其占总用例数的百分比。

为了观察存在测试用例失败时运行“make test”的结果，读者以 unitest_dll.c 文件为例修改其中的某一判定语句以故意制造失败的情形。图 28.16 示例说明了出现失败时运行“make test”的结果。注意，make 在最后报告了错误，且没有继续运行后面的单元测试可执行程序。

```
                    /embedded/buildv2
$ make test
...... 显示结果有删减 ......
Case 59 ======>
    Location: unitest_dll.c:179
    Expected: "dll_size (&list)" == "2"
 (X)  Result: Failed

Case 60 ======>
    Location: unitest_dll.c:180
    Expected: "dll_head (&list)" == "&node_tail.node_"
      Result: Succeeded

Case 61 ======>
    Location: unitest_dll.c:181
    Expected: "dll_tail (&list)" == "&node_tail.node_"
      Result: Succeeded


*************************************************
*                  Test Report                      *
*************************************************
    Total: 61
Succeeded: 60 (98%)
*************************************************


:-( Failed!

make: *** [unitest/unitest_dll.exe] Error 255
```

图 28.16

至此，单元测试无缝整合到开发环境中的工作就完成了。让我们再回顾一下其中的关键内容。首先，通过定义单元测试源程序的维护规则方便了 Makefile 的设计和单元测试源程序的维护。规则中规定，一个单元测试源文件对应于一个被测文件，且它将最终被编译成一个可以独立运行的可执行程序。基于这些规则所设计出的 Makefile，使得增加单元测试源文件很简单。

其次，调用“make unitest”可以编译出所有的单元测试可执行文件。最后，一旦单元测试可执行文件被编译出来以后，我们可以根据自己的需要手动运行它，或者运行“make test”让 make 程序逐个运行所有的单元测试可执行文件。

28.5　几个实施问题

现在让我们来审视一下单元测试在现实项目中的几个实施问题。本章介绍的单元测试方法是需要先编写单元测试代码，然后将它与被测模块所在的库进行链接以获得单元测试可执行程序的。这种方式隐含着被测模块必须不能带有 main()函数，否则将会与 unitest.h 文件中定义的 main()函数发生链接冲突。然而，每个项目一定有一个 main()入口函数，那这部分代码就没有办法用这里介绍的方法做单元测试了。为此，项目在设计的过程中可以考虑让 main()函数的实现尽可能的简单，尽量做到让 main()函数成为各模块的黏合剂，而不应存在复杂的程序逻辑。如此，我们可以干脆放弃对 main()函数进行单元测试，通过代码审查的方式就足够了。

在单元测试中难免需要触及被测模块的内部数据，以检查数据内容是否如期望的那样。那就存在这样一个问题：单元测试程序如何取得被测模块的内部数据呢？一种做法是将模块的这些数据定义为非静态的全局变量让外部直接引用。但这种做法将破坏程序的模块性，即违背了“只暴露必要的变量和函数”这一编程好习惯。

作者的解决方法是，在被测模块内定义一些用于辅助单元测试的函数，在单元测试代码中通过调用这些辅助函数获取被测模块的内部数据。采用这种方法并不妨碍将模块的内部变量定义为静态的。在辅助函数之前加上“unitest_”前缀是一个不错的编程好习惯，这可以防止它们被意外调用，更进一步，可以通过使用宏来控制这些函数的存亡。

回想前面 c.rule 的改动，其中包含当（且仅当）运行“make unitest”时，会定义 UNITEST 宏（参见图 28.8 的第 35 行）。通过在代码中使用 UNITEST 宏，可以控制辅助函数是否存在。图 28.17 示例说明了使用这个宏控制辅助函数的例子。

```
00094: static mnode_t g_mnode;
00095:
00096: ……
00097:
00098: #ifdef UNITEST
00099: mnode_t *unitest_get_mnode ()
00100: {
00101:     return &g_mnode;
00102: }
00103: #endif
```

图 28.17

对于单元测试存在这样一个误区，即一味追求高覆盖率。根据 Parasoft 公司的一份报告显示，当单元测试程序的代码覆盖率超过 70%时，团队所花费的成本就非常的高了。每一个项目应当视具体的情形去决定，哪一部分代码应当达到极高的覆盖率，哪一部分则不用太关心。一味地追求代码覆盖率有将单元测试当做救命稻草之嫌。

另外，也存在将单元测试只当做一种形式，而没有人真正关心其意义和执行效果。出现这种问题，很有可能是单元测试所带来的负担太重，或者工程师并没有意识到它的重要性。如果是负担太重，团队应当致力于找出为什么，并对单元测试流程进行改进；如果是团队没有意识到单元测试的重要性，那只能通过教导以及一定形式的带头实践去帮助大家提高认识。

28.6 桩函数和打桩

前面列举的双向链表单元测试程序有一个特点，因为双向链表是一个完全自成一体的模块，所以单元测试程序也容易写。如果一个模块需要使用其他模块的函数，或者需要调用标准库函数时，进行单元测试时麻烦会不少。下面以图 28.18 为例来说明其复杂性。

```
00095: bool foo ()
00096: {
00097:     void *p_buf = malloc (BUF_SIZE);
00098:     if (0 == p_buf) {
00099:         // logic block A
00100:         return false;
00101:     }
00102:     else {
00103:         // logic block B
00104:         return true;
00105:     }
00106: }
```

图 28.18

图中的 foo()函数调用了标准 C 库中的 malloc()函数进行内存分配，内存分配成功与否需采用不同的处理逻辑。做单元测试时，必须通过用例的设计保证逻辑块 A（第 99 行）和 B（第 103 行）都分别被执行到。设计 foo()函数单元测试用例的关键是如何让 malloc()函数在需要时返回空指针。

读者或许会想：让 malloc()函数返回空更改 foo()函数的实现不可以做到吗？比如，需要 malloc()函数返回空时，可以让 malloc()函数分配 4GB 内存，绝大部分系统会因为内存不足而使得 malloc()函数返回空。很遗憾，这种方法不行，作为单元测试，应尽可能不要去改变被测函数的实现。还有，这种貌似可行的方式很难操作。

为了解决这类问题，需要引入桩函数的概念。一个桩函数就是一个虚假的函数实现，它的行为可以根据需要从外部进行控制，以辅助完成单元测试。图 28.19 是针对 foo()函数、包含桩函数的单元测试代码，其中的 malloc()和 free()两个函数都是桩函数。请读者重点关注 malloc()

函数的实现。

```
00001: #include "unitest.h"
00002: #include "foo.h"
00003:
00004: static bool g_is_null = false;
00005:
00006: void *malloc (int _bytes)
00007: {
00008:     static char buf [BUF_SIZE];
00009:     if (!g_is_null) {
00010:        return buf;
00011:     }
00012:     return 0;
00013: }
00014:
00015: void free (void *)
00016: {
00017:     // do nothing
00018: }
00019:
00020: void unitest_main ()
00021: {
00022:     g_is_null = false;
00023:     UNITEST_EQUALS (foo (), true);
00024:     g_is_null = true;
00025:     UNITEST_EQUALS (foo (), false);
00026: }
```

图 28.19

定义 g_is_null 全局变量的目的，是为了从外部能控制 malloc()函数的行为。在 unitest_main()函数中，通过为 g_is_null 设置不同的值做到控制 malloc()函数的行为，而最终目的就是为了测试 foo()函数的实现。编写桩函数的行为被称为“打桩”。

设计了桩函数以后，需要注意相同函数实现的冲突问题。比如，foo.c 和 unitest_foo.c 在共同编译生成 unitest_foo.exe 文件时，必须保证 C 库不会被链接进去，否则会因为 unitest_foo.c 文件内存在的 malloc()和 free()函数实现，而与 C 库中的发生链接冲突。

尽管这里举的例子很简单，但对于理解桩函数和打桩行为还是很有帮助的。在现实项目中，有可能一个被测模块需要用到另外几个模块的很多函数，如果需要对所有的外部模块都进行打桩的话，工作量不可小视，这也是单元测试中非常耗费人力资源的重要一方面。另外，如何复用和管理桩函数也是一个很有学问的问题，而这需要通过一定的设计方法去解决。

28.7　错误注入，一种可测试性设计

为了完成单元测试，在大多情形下需要编写桩函数，当项目具有一定的规模时，编写桩函数所花费的努力就很可观了。如何减小单元测试所消耗的资源，值得我们从设计的角度进行考虑。接下来，我们共同探讨一种提高程序可测试性的设计方法。

我们前面看到了，单元测试最难的是构建测试所需的先决条件，且出错条件是最难构造的。在前面介绍桩函数时，我们看到了一种通过全局变量控制桩函数行为的方法。能否将这种方法不只是运用于桩函数，而是直接作为软件模块设计的一个组成部分呢？这正是后面将要介绍的错误注入这一方法的设计思想。

如果一个函数所返回的错误可以通过使用程序控制的方法，这将大大减少制造错误所花费的工作量。而控制某一个函数返回怎样错误的动作，其实就是将一个错误注入函数的行为，这一行为我们称为错误注入。

有了错误注入这一概念后，那注入的错误到底影响哪一个函数呢？这需要引入错误注入点这一概念。所谓的错误注入点，就是错误注入动作可以施加影响的某一函数内的具体位置，以下简称为注入点。注入点可以通过使用枚举的方式进行定义，图 28.20 示例说明了为定时器模块定义的三个注入点。从注入点的名称很容易猜到它们分别位于什么函数内。

```
00031: typedef enum {
00032:     INJECTION_POINT_TIMER_ALLOC,
00033:     INJECTION_POINT_TIMER_FREE,
00034:     INJECTION_POINT_TIMER_START,
00035:
00036:     // !!! NOTE: please always put the INJECTION_POINT_COUNT and
00037:     // INJECTION_POINT_LAST at the end of this enum
00038:     INJECTION_POINT_COUNT,
00039:     INJECTION_POINT_LAST = (INJECTION_POINT_COUNT - 1)
00040: } injection_point_t;
```

图 28.20

具有注入点的函数，需要在运行时检查注入点是否被注入了错误，看来我们需要创建全局变量来保存每一个注入点所注入的错误是什么。除了记录错误值，有的注入点可能还需要得到其他的数据，以进行更为特别的处理。比如，我们可能希望对定时器模块注入错误时只针对某一个名字的定时器而不是全部定时器，在这种情形下，注入错误时还得指定目标定时器的名字。定时器模块只是一个例子，每个模块都有自己特定的数据。因此，我们还得为每一个错误点保存除错误码之外的植入数据。图 28.21 示例说明了用于存储一个注入点数据所需的数据结构；另外，还示例说明了用于注入错误和获取注入错误的函数实现。

```
00030: #ifdef UNITEST
00031:
00032: typedef struct {
00033:     error_t error_;
00034:     void *p_data_;
00035: } injection_data_t;
00036:
00037: static injection_data_t g_data_array [INJECTION_POINT_COUNT];
00038:
00039: void error_inject (injection_point_t _point, error_t _error, void *_p_data)
00040: {
00041:     if (_point > INJECTION_POINT_LAST) {
```

```
00042:         return;
00043:     }
00044:
00045:     g_data_array [_point].error_ = _error;
00046:     g_data_array [_point].p_data_ = _p_data;
00047: }
00048:
00049: int injected_error_get (injection_point_t _point, void **_p_data_ptr)
00050: {
00051:     if (_point > INJECTION_POINT_LAST) {
00052:         return 0;
00053:     }
00054:
00055:     *_p_data_ptr = g_data_array [_point].p_data_;
00056:     return g_data_array [_point].error_;
00057: }
00058:
00059: #endif
```

图 28.21

第 32～35 行所定义的 injection_data_t 数据结构将被用于存储一个注入点的植入错误码和相应的数据。第 37 行定义了用于存放注入点信息的数组 g_data_array，每一个注入点占用数组中的一个元素。第 39～47 行所定义的错误注入函数 error_inject()的实现很明了，就是将注入点的信息保存到 g_data_array 数组内的对应元素中。第 49～57 行定义了 injected_error_get()函数用于获得注入点的信息。请注意，g_data_array 数组和两个函数都是在 UNITEST 宏的控制之下的，只有当 UNITEST 宏定义了的情形下它们才存在，即只有运行“make unitest”时这些函数才存在。

图 28.22 示例说明了实现 INJECTION_POINT_TIMER_START 注入点的代码。

```
embedded/code/platform/timer/v4/src/timer.c
00366: error_t timer_start (timer_handle_t _handle, msecond_t _duration,
00367:     expiry_callback_t _cb, void *_arg)
00368: {
00369:     interrupt_level_t level;
00370:
00371:     if (null == _cb) {
00372:         return ERROR_T (ERROR_TIMER_ALLOC_INVCB);
00373:     }
00374:     level = global_interrupt_disable ();
00375:     if (is_invalid_handle (_handle)) {
00376:         global_interrupt_enable (level);
00377:     return ERROR_T (ERROR_TIMER_START_INVHANDLE);
00378:     }
00379: #if UNITEST
00380: {
00381:     char *timer_name;
00382:     error_t ecode = injected_error_get (INJECTION_POINT_TIMER_START,&timer_name);
00383:
00384:     if (ecode != 0) {
00385:         if (null == timer_name) {
00386:             return ecode;
00387:         }
00388:         if (0 == strcmp (_handle->name_, timer_name)) {
00389:             return ecode;
00390:         }
```

```
00391:     }
00392: }
00393: #endif
00394:     if (TIMER_STARTED == _handle->state_) {
00395:         g_statistics.abnormal_ ++;
00396:         global_interrupt_enable (level);
00397:         return ERROR_T (ERROR_TIMER_START_INVSTATE);
00398:     }
00399:
00400:     _handle->ticks_ = _duration / CONFIG_TICK_DURATION_IN_MSEC;
00401:     if (0 == _handle->ticks_) {
00402:         _handle->ticks_ ++;
00403:     }
00404:     _handle->callback_ = _cb;
00405:     _handle->arg_ = _arg;
00406:     global_interrupt_enable (level);
00407:     timer_insert (_handle);
00408:     return 0;
00409: }
```

图 28.22

第 379～393 行是注入点的程序实现。注意，这些代码同样是在 UNITEST 宏的控制之下的。第 382 行先得到注入点的信息，包括错误码和定时器的名称。每一个注入点除了一定包含一个错误码外，所包含的其他数据完全是由每一个注入点的实现决定的。第 384 行用于查看是不是注入了错误，如果是，则 if 语句内的代码将被运行。第 385～387 行是看一看错误注入时是否提供了定时器名，如果没有说明则所注入的错误是针对所有定时器的，此时在第 386 行直接返回被注入的错误。第 388～390 行是针对被注入的错误还包含定时器名的情形，在这种情形下，timer_start()函数只是针对所指定的定时器时才返回错误。

将单元测试代码与非测试代码混在一起多少让人觉得不大舒服。因为理论上，被测代码不应包含为单元测试所编写的代码，但在较大规模的项目中要做到这一点并不是一件易事。这里所采用的方法其实是一种折中方案。

下面看一看如何通过注入错误来方便单元测试。先假设存在图 28.23 所示的被测试函数 foo()。在第 56 行，foo()函数调用 timer_start()函数启动定时器。第 57～61 行是定时器无法启动时的处理代码，其中省略了具体内容。

```
00052: error_t foo (timer_handle_t _handle)
00053: {
00054:     error_t ecode;
00055:
00056:     ecode = timer_start (_handle);
00057:     if (0 != ecode) {
00058:         // handle error
00059:         ...
00060:         return ecode;
00061:     }
00062:     ...
00063:     return 0;
00064: }
```

图 28.23

如果对 foo()函数做单元测试，则需要验证第 57～61 行的代码在执行时是否如设计所愿。图 28.24 示例说明了用于测试第 57～61 行代码的测试用例。

```
00030: void unitest_main (int argc, char *argv[])
00031: {
00032:     UNUSED (argc);
00033:     UNUSED (argv);
00034:
00035:     error_inject (INJECTION_POINT_TIMER_START,
00036:         ERROR_T (ERROR_TIMER_START_INVSTATE), null);
00037:     UNITEST_EQUALS (foo (), ERROR_T (ERROR_TIMER_START_INVSTATE));
00038:     error_inject (INJECTION_POINT_TIMER_START, 0, null);
00039:     ...
00040: }
```

图 28.24

第 35 行通过调用 error_inject()函数，为 timer_start()函数注入一个错误，所注入的错误是 ERROR_TIMER_STATE_INVSTATE，且注入的错误是针对全部定时器的。第 37 行调用 foo()函数并验证 foo()函数所返回的错误是 ERROR_TIMER_STATE_INVSTATE。第 38 行再调用 error_inject()函数清除所注入的错误。

在采用错误注入这一方法进行单元测试时，所测试的代码并不是实现注入点的函数，而是调用它的函数。比如，在前面的例子中，在 timer_start()函数中实现注入点的目的不是为了测试 timer_start()函数，而是为了测试调用 timer_start()函数的函数，即 foo()函数。

通过错误注入进行单元测试的本质，就是人为地制造错误以使得所希望的代码分支被运行。另外，由于在注入错误时可以携带参数，这在很大程度上提高了注入点所实现功能的灵活性。

错误注入这一测试方法可以进行一定的使用领域扩展，而不只局限于单元测试。现实的软件产品，最难验证的是当软件遇到错误时，是否仍能对之处理并继续提供其他的服务而不是崩溃。因此，可以通过错误注入的方式，人为地制造一定的错误以查看我们所设计的软件是否能有效地处理它。如果将错误注入功能做成一个可以通过命令行进行控制的功能，那么测试人员可以通过命令行来选择需要注入的错误，从而提高测试效率。

可测试性设计的另一种内涵是，软件的设计应使自动化测试更加便利。比如，存在图形界面的设备就难以将测试自动化，因为图形界面的存在需要人眼加以识别来判断测试成功与否。一种提高可测试性的方法是，在软件中针对每一个图形界面及界面中的动作生成一定的数字标识，这种标识有点像 15.1 节介绍的错误码。如果软件设计成当图形界面发生变化时，在串口（也可以是其他的接口）上同步输出其标识，测试软件就可以通过标识判断测试是否成功而省去人的参与，从而使得自动化测试成为可能。

可测试性设计的考虑有助于提高测试效率和效果，它值得我们像对待用户需求那样重视。

28.8 平台开发与单元测试

单元测试应当尽可能放在开发主机上完成，而不是在目标机上。这样做有一个好处，就是不会因为目标机资源的不足而使得单元测试出现瓶颈。另外，在开发主机上做单元测试也更加方便，可以避免测试程序向目标机上传送。

要让嵌入式软件能在开发主机上进行单元测试，需要使用到平台开发技术。对于前面列举的双向链表模块，由于它不存在对其他模块的依赖，因此不存在平台依赖问题。但是，对于一些依赖操作系统功能的模块，如果希望在开发主机上进行单元测试，则必须先设计一个跨平台的库，之后，构建在跨平台的库上的模块就能在开发主机上进行单元测试了。

在打造平台时可以考虑运用上一节介绍的可测试性设计方法，使得构建于平台上的模块能更方便地完成单元测试。

如果在开发主机环境中不具备一些嵌入式设备中所特有的资源，那又如何在开发主机上进行单元测试呢？有以下几种选择。首选方式：平台通过封装的方法提供访问资源的函数，以及在开发主机上模拟真实资源的行为。如果这种方式不可行或过于复杂，那么可以采取将单元测试放到嵌入式设备上去完成的方式。再不然，就只能放弃对它做单元测试了。

如果被测模块与嵌入式设备中的硬件相关，对这些模块做单元测试会相对麻烦。比如，很多嵌入式操作系统并不像 Linux 或 Windows 操作系统那样，可以随时加载一个可执行程序；相反，操作系统与应用是被编译在同一个可执行程序中的，对于这种情形的单元测试，如果要采用本书介绍的 Makefile 进行编译，应将模块目录名放入 embedded/buildv2/Makefile 的 NONUT_DIR 变量中，且用“make release”或“make debug”命令来编译单元测试可执行程序。使用这种方式，会出现无法统计代码覆盖率这种情形。

从提高项目单元测试效率的角度来看，项目中的代码应尽可能多地被设计成能在开发主机上进行单元测试。因此，与嵌入式设备资源相关的代码应尽量封装到平台（库）中，这是平台开发工作需要注意的一个关键点。在开发主机上做单元测试还有一个好处是：可以运用代码覆盖工具统计代码覆盖率，以及使用动态检查工具实现对代码的动态检查。一旦将单元测试放到嵌入式设备上进行，由于这些工具可能无法在嵌入式设备上运行而使得失去一些保证质量的机会。

28.9 被测行为的确定性

单元测试在开发主机上完成后，是否还需要在嵌入式设备上再进行呢？答案并不是简单的“是”或“不是”，而是视情况而定。

在做进一步的讨论之前，需要了解被测模块的动态和静态行为。动态行为是指被测模块的

行为与操作系统的实现紧密相关，且存在因操作系统不同而使得代码的行为表现也不同的可能。比如，平台中的线程和套接字封装模块的行为就应当是动态的。与动态行为不同的是，静态行为则是指被测模块无论在什么操作系统上它的行为表现始终如一，或者说这类模块根本不依赖操作系统。比如，前面列举的双向链表模块，它的行为就是静态的。

有了动态和静态行为的概念后，就很容易明白哪些模块既要在嵌入式设备上进行单元测试，又要在开发主机上进行单元测试。其原则是：对于只具有静态行为的模块在开发主机上进行单元测试就行了，而对于具有动态行为的模块必须在嵌入式设备和开发主机上同时做单元测试。或者说，因为具有静态行为的模块其行为具有确定性，所以只要在开发主机上测试就行了。

当一个平台提供对 Linux 和 VxWorks 两个操作系统的线程或任务封装时，显然，针对不同的操作系统封装模块所调用的函数也不同，而对这一封装在各操作系统上进行单元测试进行验证就显得理所当然了。即使是同一类操作系统，比如桌面的 Linux 和嵌入式设备中的实时 Linux，对具有动态行为的模块在两个操作系统上分别进行单元测试也仍有必要。

对具有动态行为的模块进行单元测试，其范畴是否仍属于单元测试？作者的回答是：是。因为它仍属于白盒测试。另外，抛开范畴之争，单元测试的目的到底是什么？是为了保证被测代码行为的正确性！既然如此，那只要所做的行为能验证程序的正确性就应是根本，而不应过于纠结于它是“黑”还是“白”。

28.10　测试用例的有效性

在一次与作者的同事探讨用例设计的有效性时，他指出模块的未定义行为这一概念，这一概念让作者一下子想到了在做嵌入式软件开发时，硬件手册中经常会出现“对于 XXX 寄存器的位 0 必须设置成 0，设置成 1 的行为没有定义”这类陈述。相似地，在单元测试中也需要关注被测模块的未定义行为，因为这涉及了测试用例的有效性和资源的合理使用。

图 28.25 是来自双向链表模块 dll_push_head()函数的具体实现。如果单是从单元测试的角度来看，当参数_p_dll 和_p_node 中存在空指针时，这一函数一定会造成程序崩溃。那设计输入空指针的测试用例合理吗？

```
void dll_push_head (dll_t *_p_dll, dll_node_t *_p_node)
{
    if (0 == _p_dll->head_) {
        _p_dll->head_ = _p_dll->tail_ = _p_node;
        _p_node->next_ = _p_node->prev_ = 0;
    }
    else {
        _p_node->next_ = _p_dll->head_;
        _p_node->prev_ = 0;
        _p_dll->head_->prev_ = _p_node;
```

```
        _p_dll->head_ = _p_node;
    }

    _p_dll->count_ ++;
}
```

图 28.25

在 27.1.7 节介绍“在接口上防范错误”这一编程好习惯时指出，对于参数的有效性检查应当以模块为边界，而对于模块内部的子模块可以考虑不进行输入参数有效性检查。另外还指出，有些模块为了效率并不对其输入参数进行有效性检查，而是将保证输入参数有效性的责任交给了函数的调用者。dll_push_head()函数的实现，正是将保证输入参数有效性的责任交给了使用者，在这种情形下，如果设计一个用空指针传入 dll_push_head()函数的测试用例就不合理了。

从 dll_push_head()函数的实现来看，当传入指针为空时是一种“未定义行为”。我们可以理解为该函数存在这样的陈述：输入参数的有效性是由函数调用者来保证的，当所传入的参数无效时其行为是未定义的。作者的观点是，在单元测试中测试模块的未定义行为是不合理的，为了节约资源我们应当避免这种行为。

被测模块未定义行为的提出虽然有助于判定单元测试用例的有效性，但也带来了另一个困境：写单元测试的人如何知道哪些行为是未定义的呢？由此看来，单元测试用例的设计最好是被测模块的设计者本人，因为他最清楚哪些模块的行为是未定义的。

未定义行为有时也可以理解为：这种行为在现实中不会发生。

28.11 小结

在项目中部署单元测试，不光时机很重要，而且还应将其无缝整合到开发环境中以提高易用性。

单元测试的部署具有一定的系统性，不论是时机、平台开发还是模块化设计都与其密切相关。我们应避免孤立地看待单元测试及其部署。



第 29 章 代码覆盖，单元测试效果的衡量指标

判定单元测试的效果有一个专门的指标，那就是代码覆盖率，它是指被测试代码占代码总数的比率[①]。我们很容易认为覆盖率越高则代码的验证程度就越高。但这一观点成立的前提是所获得的覆盖报告是以合理测试用例为前提而获得的。现实项目中存在这种情形，即为了获得百分百的覆盖报告而不顾用例的合理性，这种情形下获得的覆盖率并不能说明代码的被测试程度。通过测试用例的设计，我们希望对被测模块获得百分百的代码覆盖率，但前面提到了，为了实现这一目标所需付出的努力极大。因此，我们应根据团队和项目的情况设置一个合适的覆盖率，而不能一味地追求百分百。

通过代码覆盖工具能获得的不只是代码覆盖率，所产生的代码覆盖报告还包含其他信息。比如，通过代码覆盖报告，我们可以知道哪些代码被运行过了，哪些没有；也可以了解运行了的各代码行各行被执行了多少次；等等。很显然，这样的报告对于单元测试用例的设计具有指导意义。在设计单元测试用例时，我们可以根据代码覆盖报告了解哪些代码还没有被测试到，进而可以有针对性地设计测试用例。

对于代码覆盖工具我们还得感谢 GNU，它不光为我们带来了被广泛使用的 gcc/g++编译器，而且还带来了代码覆盖工具——gcov，且这一工具与 gcc/g++可以实现无缝结合。除了 gcov，还有另一个有用的开源工具——lcov，它可以用于获得更具可读性、HTML 格式的代码覆盖报告。

尽管 gcov 与 gcc/g++可以无缝结合，但是，对于我们更为有用的是需要将 gcov 和 lcov 像单元测试那样无缝整合到开发环境中以提高其可使用性。如果在做完单元测试以后，可以在开发环境中运行“make creport”而获得代码覆盖报告那就太好不过了。这正是本章最终要实现的目标。

① 正因为代码覆盖是衡量单元测试的指标，所以，单元测试用例的数量并不重要。

29.1 了解代码覆盖工具

只要读者的开发环境中有 gcc 编译器，那就应当同时获得了 gcov，因为 gcov 与 gcc 是一起发布的。图 29.1 示例说明了如何检查 gcov 在开发环境中是否存在。

```
$ gcov --version
gcov (GCC) 4.3.4 20090804 (release) 1
Copyright (C) 2008 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.
```

图 29.1

至于 lcov，读者可以从官网（http://ltp.sourceforge.net/coverage/lcov.php）上下载（本书光盘中也有），然后按图 29.2 所示的那样安装并验证是否安装成功。由于 lcov 是一个采用 Perl 语言编写的工具，读者必须保证自己的开发环境安装了 Perl 语言。

```
$ tar xzf lcov-1.9.tar.gz
                          /embedded/buildv3
$ cd lcov-1.9
                          /embedded/buildv3
$ make install
bin/install.sh bin/lcov /usr/bin/lcov -m 755
bin/install.sh bin/genhtml /usr/bin/genhtml -m 755
bin/install.sh bin/geninfo /usr/bin/geninfo -m 755
bin/install.sh bin/genpng /usr/bin/genpng -m 755
bin/install.sh bin/gendesc /usr/bin/gendesc -m 755
bin/install.sh man/lcov.1 /usr/share/man/man1/lcov.1 -m 644
bin/install.sh man/genhtml.1 /usr/share/man/man1/genhtml.1 -m 644
bin/install.sh man/geninfo.1 /usr/share/man/man1/geninfo.1 -m 644
bin/install.sh man/genpng.1 /usr/share/man/man1/genpng.1 -m 644
bin/install.sh man/gendesc.1 /usr/share/man/man1/gendesc.1 -m 644
bin/install.sh man/lcovrc.5 /usr/share/man/man5/lcovrc.5 -m 644
bin/install.sh lcovrc /etc/lcovrc -m 644
                          /embedded/buildv3
$ lcov --version
lcov: LCOV version 1.9
```

图 29.2

要了解 gcov 是如何帮助我们检查代码覆盖的，可以从图 29.3 所示的简单程序开始。

```
00001: #include <stdio.h>
00002:
00003: void foo (int _option)
00004: {
00005:    if (0 == _option) {
00006:         printf ("option = 0\n");
00007:    }
00008:    else {
00009:         printf ("option != 0\n");
00010:    }
00011: }
```

```
00012:
00013: int main ()
00014: {
00015:     foo (0);
00016:     foo (1);
00017:     return 0;
00018: }
```

图 29.3

图 29.4 示例说明了如何编译 foo.c 以获得文本格式的代码覆盖报告。

```
                         /coverage
$ gcc -gdwarf-2 -g3 -fprofile-arcs -ftest-coverage foo.c -o foo.exe
                         /coverage
$ ls
foo.exe*  foo.c  foo.gcno
                         /coverage
$ ./foo.exe
option = 0
option != 0
                         /coverage
$ ls
foo.exe*  foo.c  foo.gcda  foo.gcno
                         /coverage
$ gcov foo.c
File '/usr/include/stdio.h'
Lines executed:0.00% of 8
/usr/include/stdio.h:creating 'stdio.h.gcov'

File 'foo.c'
Lines executed:100.00% of 9
main.c:creating 'foo.c.gcov'
                         /coverage
$ ls
foo.exe*  foo.c  foo.c.gcov  foo.gcda  foo.gcno  stdio.h.gcov
                         /coverage
$ cat main.c.gcov
        -:    0:Source:main.c
        -:    0:Graph:main.gcno
        -:    0:Data:main.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:void foo (int  option)
        2:    4:{
        2:    5:    if (0 ==  option) {
        1:    6:        printf ("option = 0\n");
        -:    7:    }
        -:    8:    else {
        1:    9:        printf ("option != 0\n");
        -:   10:    }
        2:   11:}
        -:   12:
        -:   13:int main ()
        1:   14:{
        1:   15:    foo (0);
        1:   16:    foo (1);
        1:   17:    return 0;
        -:   18:}
```

图 29.4

为了获得代码覆盖信息，在编译源文件时需要使用选项-fprofile-arcs和-ftest- coverage。使用这两个选项时，编译器会为每一个源文件生成一个后缀为.gcno 的同前缀名文件。从图 29.4 可以看到，foo.c 对应的编译生成文件是 foo.gcno 文件。

编译完了后运行生成的 foo.exe 文件。如果检查文件目录，读者会发现生成了一个新的文件——foo.gcda。请注意，.gcda 文件只有当测试程序退出时才会生成。如果希望程序不退出也生成.gcda 文件，则需要在程序中调用 gcov 所提供的函数。具体的函数名读者可以查看相关资料，在本书我们并不需要用到这一功能。

一旦生成.gcda 文件，就可以运行 gcov 以获得代码覆盖报告。在本例中，生成了 stdio.h.gcov 和 foo.c.gcov 两个文件。通过“cat foo.c.gcov”命令，读者可以看到左边的数字指示了各行代码被运行的次数，具体含义不打算细讲，因为使用 lcov 工具所获得的报告更具可读性。接下来让我们看一看如何使用 lcov。

使用 lcov 生成更具可读性的报告分两步。第一步，使用 lcov 命令生成一个中间文件，如图 29.5 所示。

```
/coverage
$ lcov -c -d . -b . >> unitest.report
Capturing coverage data from .
Found gcov version: 4.3.4
Scanning . for .gcda files ...
Found 1 data files in .
Processing foo.gcda
Finished .info-file creation
```

图 29.5

第二步，以 lcov 所生成的 unitest.report 文件作为另一个 lcov 自带工具 genhtml 的输入文件，生成 HTML 格式的覆盖报告。图 29.6 示例说明了如何使用 genhtml 工具。

```
/coverage
$ genhtml unitest.report
genhtml unitest.report
Reading data file unitest.report
Found 3 entries.
No common filename prefix found!
Writing .css and .png files.
Generating output.
Processing file /coverage/foo.c
Processing file /usr/include/stdio.h
Processing file /usr/include/byteswap.h
Writing directory view page.
Overall coverage rate:
  lines......: [illegible] (9 of [illegible] lines)
  functions..: 33.3  (2 of 6 functions)
  branches...: 16.7  (2 of 12 branches)
```

图 29.6

一旦 genhtml 运行之后，在工作目录下将生成 index.html 等文件。通过使用网页浏览器打开这个文件可以获得整个项目（当然现在的例子只有 foo.c 文件）的代码覆盖报告，所有源文

件的组织形式与目录结构是相似的。图 29.7 所示是 foo.c 文件的代码覆盖报告。从图中右上角可以看到，在代码行、函数和分支三个纬度上，foo.c 代码的测试率都达到了百分百。

LCOV - code coverage report

Current view: top level - coverage - foo.c (source / functions)
Test: unitest.report
Date: 2011-01-03

	Hit	Total	Coverage
Lines:	9	9	100.0 %
Functions:	2	2	100.0 %
Branches:	2	2	100.0 %

```
      Branch data     Line data    Source code
 1               :             : #include <stdio.h>
 2               :             :
 3               :             : void foo (int _option)
 4               :           2 : {
 5   [ +  + ]:           2 :         if (0 == _option) {
 6               :           1 :                 printf ("option = 0\n");
 7               :             :         }
 8               :             :         else {
 9               :           1 :                 printf ("option != 0\n");
10               :             :         }
11               :           2 : }
12               :             :
13               :             : int main ()
14               :           1 : {
15               :           1 :         foo (0);
16               :           1 :         foo (1);
17               :           1 :         return 0;
18               :             : }
```

Generated by: LCOV version 1.9

图 29.7

读者或许会奇怪，为什么图 29.6 最终所显示的覆盖率却远远低于图 29.7 所示的百分比呢？注意观察图 29.6 的输出信息将发现，对应的统计百分比是将标准库中的 stdio.h 和 byteswap.h 两个文件包含在内的。

很显然，在统计代码覆盖率时，我们并不关心标准库中的文件。通过 lcov 的-r 选项，可以过滤掉那些我们不希望的文件。图 29.8 示例说明了如何过滤掉/usr/include 目录下的所有文件，并生成新的中间文件 clear.report。从命令的运行结果可以看出，三个纬度的百分比都达到了一百。

```
                /coverage
$ lcov -r unitest.report /usr/include/* -o clear.report
Reading tracefile unitest.report
Removing /usr/include/byteswap.h
Removing /usr/include/stdio.h
Deleted 2 files
Writing data to clear.report
Overall coverage rate:
  lines......: 100.0  (9 of 9 lines)
  functions..: 100.0  (2 of 2 functions)
  branches...: 100.0  (2 of 2 branches)
```

图 29.8

如果覆盖率不是百分百，所显示的代码覆盖报告又是怎样的呢？我们还是基于 foo.c 文件做一个试验。在新的试验中，需要先删除图 29.3 中的第 16 行代码。图 29.9 显示了最终的代码覆盖报告。注意，在再一次获得 foo.c 文件的代码覆盖报告之前，必须先清除所有上次生成的文件，其中最重要的是.gcna 文件。如果不做这一步，将出现生成的代码覆盖报告与之前的是一样的。作者猜测出现这一现象，是因为.gcna 一旦生成后，其中的统计信息并不会因为我们重新获得覆盖报告而清除，这也很好理解，因为.gcna 可以记录多个可执行文件中所包含同一文件的代码覆盖信息。

从图 29.9 看来，由于调用 foo()函数时少了输入参数为 1 的情形，所以造成有一个分支的代码没有被运行到，因此最终影响了代码行和分支的覆盖率。

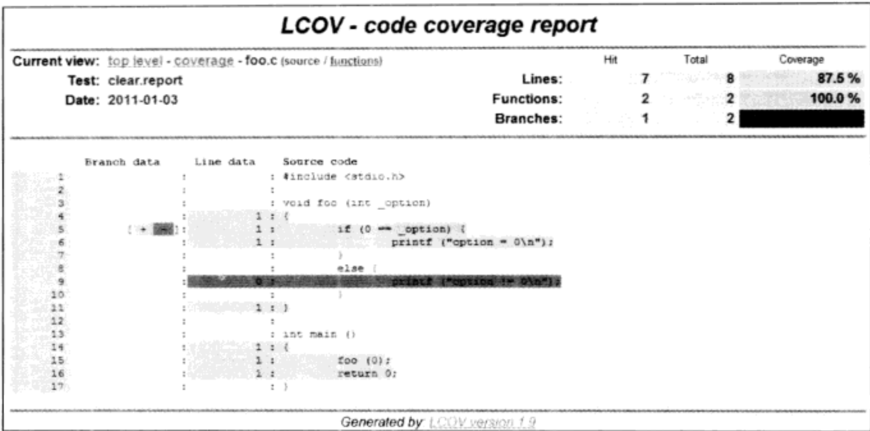


图 29.9

如果想使用 lcov 生成 HTML 格式的代码覆盖报告，我们并不需要直接使用 gcov 工具，这个工具将会被 lcov 工具在幕后使用。前面使用 gcov 的目的完全是为了让读者能直观地看到 gcov 工具的效用。

29.2　无缝整合代码覆盖

要获得一个代码覆盖报告，需要如下几个步骤。

（1）源程序在编译时需要使用“-fprofile-arcs”和“-ftest-coverage”选项。

（2）执行可执行程序以获得分析所需的.gcda 文件。

（3）运行 lcov 工具（集）以分析.gcda 文件并生成最终的 HTML 格式的报告。

那如何将 gcov 和 lcov 集成到开发环境中呢？在图 26.3 中指出，代码覆盖应以单元测试为中心。在讲解单元测试时谈到进行单元测试需要分两步，即“make unitest”和“make test”。能否复用这两步，再加上第三步以最后生成代码覆盖报告呢？这绝对是一个好主意！这种方法也正体现了以单元测试为中心，通过进行单元测试可以“顺便”得到代码覆盖报告。

29.2.1　更改 Makefile

下面我们就沿着复用“make unitest”和“make test”两步的思路来改进 Makefile，并新增“make creport”以获得单元测试后的代码覆盖报告。

第一个改动点位于 c.rule 中，即当构建 unitest 目标时，需要增加“-fprofile- arcs”和“-ftest-

coverage”两个编译选项用于编译源程序，改动如图 29.10 中的第 33 行所示。

```
00028: ……
00029: ifeq ("$(MAKECMDGOALS)", "unitest")
00030: DIR_OBJS = uobjs
00031: AR = ar
00032: CC = gcc
00033: CFLAGS += -gdwarf-2 -g3 -DUNITEST -fprofile-arcs -ftest-coverage
00034: LINK_LIBS += gcov
00035: endif
00036: ……
```

图 29.10

图 29.10 中还增加了第 34 行。这是因为使用 gcov 需要将 libgcov.a 库链接到可执行程序中。后面的改动全部位于 embedded/buildv3/Makefile 文件中，如图 29.11 所示。

```
00071: ……
00072: ifeq ("$(MAKECMDGOALS)", "creport")
00073: MAKE_DIRS += $(UT_DIRS)
00074: endif
00075:
00076: RM = rm
00077: MKDIR = mkdir
00078: RMFLAGS = -fr
00079: RMS = release debug unitest $(DIR_COVERAGE)
00080:
00081: DIR_UNITEST = unitest
00082: DIR_COVERAGE = coverage
00083:
00084: .PHONY: release debug clean touch unitest test force creport
00089: ……
00119:
00120: REPORT = unitest.report
00121: TEMP = temp.report
00122:
00123: ifeq ("$(MAKECMDGOALS)", "creport")
00124: ifeq ("$(wildcard $(DIR_UNITEST)/tested)", "")
00125: $(error Did you forget to run 'make unitest' and 'make test'?)
00126: endif
00127: endif
00128:
00129: $(DIR_COVERAGE):
00130:     $(MKDIR) $@
00131:
00132: creport: $(DIR_COVERAGE)
00133:     @set -e; \
00134:     cd $(DIR_COVERAGE); \
00135:     $(RM) $(RMFLAGS) *; \
00136:     for DIR in $(MAKE_DIRS); \
00137:        do \
00138:           if ls $$DIR/uobjs/*.gcda > /dev/null 2>&1; then \
00139:              lcov -c -d $$DIR/uobjs -b $$DIR >> $(TEMP); \
00140:           fi; \
00141:        done ; \
00142:     lcov -r $(TEMP) /usr/include/* tests/* tools/* c++/*
```

```
                unitest.h errstr.def -o $(REPORT); \
00143:      genhtml $(REPORT); \
00144:      $(RM) $(RMFLAGS) $(TEMP) $(REPORT)
00145:      @echo ""
00146:      @echo ":-) Generated"
00147:      @echo ""
```

图 29.11

其中的更改包括：

（1）第 72～74 行，在构建 creport 目标时需要让 MAKE_DIRS 变量包含 UT_DIRS 变量中的目录。

（2）第 77 行定义了 MKDIR 变量，用于记录目录创建命令的命令名称，即“mkdir”。后面需要使用该命令在 buildv3 目录下创建新目录，用于存放代码覆盖报告。

（3）第 79 行则将代码覆盖报告的目录加入到 RMS 变量中，实现运行“make clean”时一并将已生成的代码覆盖报告删除。

（4）第 82 行定义了 DIR_COVERAGE 变量保存代码覆盖报告的存放目录名，且将其值设置为“coverage”。当构建 creport 目标时会在 buildv3 目录下新建这一目录。

（5）第 84 行将 creport 定义为一个假目标。creport 源于“coverage report”的简写形式。

（6）第 120 和 121 行定义了两个新的变量，其中放置的是生成代码覆盖报告时中间文件的名称。

（7）第 123～127 行判断 unitest 目录下的 tested 文件是否存在，这是为了可使用性。在生成代码覆盖报告之前，我们必须保证进行过单元测试，因为只有这样才能生成代码覆盖报告所需的.gcda 文件。

（8）第 129～130 行所新增的规则用于在 buildv3 目录下创建 coverage 目录。

（9）132～147 行所新增的规则是用于生成代码覆盖报告的。在第 132 行，让 creport 目标依赖于$(DIR_COVERAGE)，这将导致在运行规则的命令之前先创建 coverage 目录。第 134 行进入 coverage 目录，接着在第 135 行将 coverage 目录中的所有文件都先删除。第 136～141 行则进入每一个列在 MAKE_DIRS 变量中的目录，以便使用 lcov 生成中间文件。第 138 行先检查所进入目录下的 uobjs 子目录内是否存在.gcda 文件，如果有才调用第 139 行的 lcov 进行分析，这有助于节约报告生成时间。第 142 行则运用 lcov 的“-r”选项过滤掉那些我们不希望统计的文件或目录，过滤完的中间信息将放入 unitest.report 文件中。第 143 行调用 genhtml 工具生成 HTML 格式的代码覆盖报告。在第 144 行则删除中间生成的临时文件。

29.2.2 检查整合效果

要生成代码覆盖报告需要通过三次不同的目标构建，分别是“make unitest”、“make test”

和“make creport”。当这三次构建动作完成了以后，在 buildv3 目录下将生成 coverage 目录，且在 coverage 目录中将看到 HTML 格式的代码覆盖报告。图 29.12 示例说明了运行结果。

```
/embedded/buildv3
$ ls
Makefile*  cxx.rule*  c.rule*  coverage/  unitest/
/embedded/buildv3
$ ls coverage
amber.png     board/             device/              emerald.png
glass.png     index-sort-f.html  index.html           snow.png
task/         updown.png         arch/                common/
driver/       gcov.css           index-sort-b.html    index-sort-l.html
ruby.png      sync/              timer/
```

图 29.12

将其中的 index.html 用网页浏览器打开以后就可以查看代码的覆盖情况。图 29.13 所示是所获得的项目整体概况。通过点击其中的链接，可以查看每一个运行到的文件的代码覆盖情况，图 29.14 展示了 dll.c 代码覆盖状况的一个片段。

LCOV - code coverage report

Current view:	top level		Hit	Total	Coverage
Test:	unitest.report	Lines:	1253	2107	
Date:	2011-01-04	Functions:	133	197	
		Branches:	401	857	

Directory	Line Coverage			Functions		Branches	
arch/x86/simulator/inc							
arch/x86/simulator/src		91.6 %	174 / 190	93.9 %	31 / 33		
board/simulator/src							
common/inc						-	0 / 0
common/src							
device/src				90.0 %	9 / 10		
driver/ctrlc/src							
sync/v3/src							
task/v3/src				76.1 %	35 / 46		
timer/v3/src							

Generated by: LCOV version 1.9

图 29.13

请注意，在下一次获得代码覆盖报告之前必须先进行一次“make clean”操作，这是因为.gcda 文件具有累积效应。这一额外的“make clean”动作将降低单元测试的效率，缓解的方法是可以考虑修改 embedded/buildv3/Makefile，将无关的模块从 COMMON_DIRS 和 UT_DIRS 两个变量中注释掉，即减少每一次构建的文件数量。这种方法之所以可行，是因为单元测试本来就是基于模块的。

29.3　三个代码覆盖程度指标

相信读者从前面的代码覆盖报告中注意到了 gcov 所使用的三个覆盖程度指标，它们分别是行覆盖率、函数覆盖率和分支覆盖率。现在，我们来看一看这三个指标的具体区别。

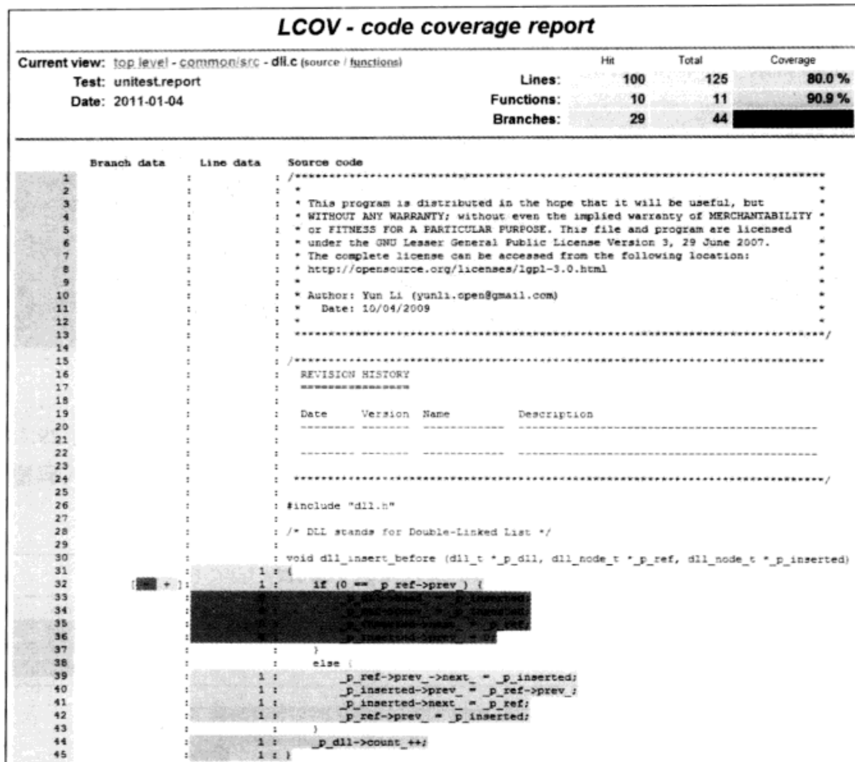


图 29.14

行覆盖是从代码行的角度度量被运行代码的覆盖程度，将运行到的代码行数与总行数相除就得到了行覆盖率。相似地，函数覆盖是度量函数被运行的程度。分支覆盖是度量条件分支（像 if、switch 这样的语句）被运行到的程度。很显然，函数覆盖的粒度是最低的，百分百的行覆盖率一定意味着百分百的函数覆盖率，但却不一定是百分百的分支覆盖率。

在这三个覆盖率中，分支覆盖率的提高可能需要花费比其他覆盖率更大的努力。

29.4 小结

单元测试的效果可以通过代码覆盖来衡量。通过代码覆盖报告，可以有效地帮助我们发现哪些代码没有被测试到，进而指引测试用例的设计方向。

本章介绍了如何将 gcov 和 lcov 无缝集成到开发环境中，使得它可以被方便地用于开发活动中。

第 30 章 静态分析，防止将失误带给用户

C 编程语言绝对不像 Java 编程语言那样优雅，其中存在不少语义陷阱，这些陷阱有可能造成所编写的程序出现预想不到的缺陷。为了提高代码质量，在编码阶段进行严格的语义检查就显得很有必要了。

早期为了简化编译器的实现，编译器并没有对代码做很细致的语义分析，而是将这一工作独立出来放到了另一个程序（工具）中去做，这就是 lint 工具。lint 的功能可以理解成是对代码进行一次“编译”，只不过“编译”过程并不生成目标文件，而只是对程序语义做更为严格的检查。lint 的这种行为就被称为静态分析，其中的静态是指对代码的分析并不需要运行被开发的程序。现在的编译器有一种发展趋势，将交给 lint 的工作更多地纳入到自身范畴，也就是在编译时也进行更为严格的语义检查。

代码审查是常用的用于保证代码编写质量的方法，而结对编程在敏捷开发方法大行其道的今天也深受推崇，有了这些方法以后还需要代码静态分析吗？当然需要！不论是代码审查也好，还是结对编程也好，其主体是人，它需要参与者在进行这些工作时全神贯注，以及要求参与者具有一定的经验。如果这些条件不能满足，其效果将大打折扣。与之不同的是，代码静态分析的主体是工具，工具不存在“走神”问题，其总是忠实如一、高质量地完成每一次检查工作。

静态分析工具的眼中只有编程语言的语法和语义，其他的内容它都一无所知，因此不要寄希望于它能帮助找到语法和语义之外的任何问题。

30.1 认识静态分析工具

pc-lint 是由 Gimpel 公司开发的一款运行于 Windows 操作系统之上的静态分析工具，另一个 Linux/UNIX 版本的工具名称是 FlexLint。两个版本的功能是一样的，但收费却不同，因为 FlexLint 是采用提供源代码的方式进行发布的。除了 Gimpel 公司的 pc-lint/FlexLint 外，还有其他的公司出品了各自的静态分析工具，比如 Klocwork 公司。在这里我们将以 pc-lint 为例进行讲解，选择它的原因是：

（1）pc-lint 用起来非常的简单，很适合与开发环境进行整合。一个工具如果要好用，必须与开发环境进行无逢整合，这一点本书反复强调。如果进行代码静态分析简单到只需运行一下“make scheck”，大家一定会乐于使用。有些静态分析工具采用的是客户/服务器模型，这种软件运用起来稍微复杂一点，有将工具复杂化之嫌。

（2）为了消除代码中的一个不是问题的错误或警告，pc-lint 可以通过使用在代码中加注释的方式进行抑制，这种方式用起来很简便。

（3）另一个重点是价格。pc-lint 比 FlexLint 便宜，而比其他的工具更便宜。官网上单用户 pc-lint 的价格是 389 美元左右，与其他动辄上万乃至十万的工具相比，pc-lint 的价格能被很多公司所接受。

pc-lint 可以在 Cygwin 环境中直接使用，如果希望它也被运用于 Linux 操作系统中，就需要借助一个在 Linux 操作系统上模拟 Windows 环境的开源项目——Wine。如何在 Linux 操作系统内配置 Wine 让 pc-lint 能工作不打算在此介绍，请读者自行上网搜索解决方案。

由于 pc-lint 是一个商用软件，因此本书的光盘中并不带有这个工具；如果读者手上有这个工具，则只需将 lint-nt.exe 拷贝到 embedded/tools/lint 目录下就行了（其他的文件一概不用拷贝）。

拷贝了 lint-nt.exe 之后，请先检查 pc-lint 的版本信息，作者所使用的版本是 8.00x，如图 30.1 所示。如果读者所使用的版本与本书的不匹配，在对本书附带的代码进行静态分析时，可能还会出现错误或警告。

```
/embedded/tools/lint
$ ./lint-nt -v
PC-lint for C/C++ (NT) Vers. 8.00x, Copyright Gimpel Software 1985-2008
```

图 30.1

还有一点需要注意，pc-lint 的配置文件 co-gnu3.lnt 与具体的版本可能有关，书中光盘所带的这个文件也是与 8.00x 相匹配的。书中所提供的源代码都采用 8.00x 版本的 pc-lint 进行过静态分析，而且没有发现错误和警告。

如果读者的 pc-lint 版本比本书所使用的低且也是 8.0 的，则可以考虑到 Gimple 的官网上下载补丁进行升级，相关的升级程序可以从 http://www.gimpel.com/html/ ptch80.htm 上下载。其中需要下载的程序包括打补丁的程序 patch.zip 和补丁文件。打补丁的方法在网页中有清楚的说明，读者照做就行了。

pc-lint（对应的可执行文件为 lint-nt.exe）需要通过命令选项来告诉它所应掌握的必要信息才能完成对代码的静态分析工作，为了简化用户接口，pc-lint 还提供另外一种方式来获取设置信息，即.lnt 文件。我们可以将 pc-lint 的命令选项放入文件中，并在运行它时指定该文件的方式。图 30.2 示例说明了 std.lnt 文件中的内容。从图中可以看出，如果需要在一个.lnt 文件中包含另一个.lnt 文件，则只要将被包含的文件名放入其中就行了。-si4 和-sp4 是两个设置选项，其

所起到的效果与“lint-nt.exe -si4 -sp4”命令行是一样的。

```
00001: //  Gnu C/C++ (version 2.95.3 or later), -si4 -sp4, lib-stl.lnt
00002: //  Standard lint options
00003:
00004: au-sm123.lnt
00005: co-gnu3.lnt
00006: options.lnt
00007:
00008: -si4 -sp4
```

图 30.2

在 embedded/tools/lint 目录下存在四个配置文件，分别是 std.lnt、au-sm123. lnt、co-gnu3.lnt 和 options.lnt。au-sm123.lnt 和 co-gnu3.lnt 是 pc-lint 工具自带的文件，au-sm123.lnt 中定义了与《Effective C++》第三版相匹配所需的检查选项，而 co-gnu3.lnt 则定义了 gcc/g++所需的选项，读者可以打开两个文件看一看以了解大概内容。std.lnt 的作用应当不用多说，它是使用 pc-lint 时唯一需要出现在命令行上的配置文件，而 options.lnt 则需要根据具体的项目录入不同的选项，可以认为 options.lnt 是针对每一个项目需要我们定制的。

在使用 pc-lint 之前，需要根据 Cygwin 环境设置 options.lnt 文件，所进行的设置内容主要是告诉 pc-lint，Cygwin 环境中 C/C++语言库的头文件（后面简称为系统头文件）的路径是什么。图 30.3 示例说明了在作者的 Cgywin 环境中所需的设置，注意其中的第 13～18 行就是告诉 pc-lint 系统头文件的具体路径。

```
00001: // Please note -- this is a representative set of error
00002: // suppression options. Please adjust to suit your own
00003: // policies See manual (chapter LIVING WITH LINT)
00004: // for further details.
00005:
00006: // unit checkout
00007: -u
00008:
00009: -d__CYGWIN__
00010: -d_HAVE_STDC
00011:
00012: // libraries for your environment
00013: -IC:\cygwin\lib\gcc\i686-pc-cygwin\4.3.4\include\c++
00014: -IC:\cygwin\lib\gcc\i686-pc-cygwin\4.3.4\include\c++\i686-pc-cygwin
00015: -IC:\cygwin\lib\gcc\i686-pc-cygwin\4.3.4\include\c++\backward
00016: -IC:\cygwin\lib\gcc\i686-pc-cygwin\4.3.4\include
00017: -IC:\cygwin\lib\gcc\i686-pc-cygwin\4.3.4\include-fixed
00018: -IC:\cygwin\usr\include
00019:
00020: -wlib(0)
00021:
00022: -e746
00023: // disable the macro warning for unrefered and unused
00024: -e755 -e757
00025: // old style comment
00026: -e1904
```

```
00027: // macro could become const variable
00028: -e1923
00029: // constructor has private access specification
00030: -e1704
00031: -e537
00032: // Use of goto is deprecated
00033: -e801
00034: -e952
00035: -e953
00036: -e954
00037: -e1932
```

图 30.3

如何知道 Cygwin 中系统头文件在 Windows 操作系统上的具体路径呢？这需要借助 gcc 和 Cygwin 环境中的 cygpath 工具。通过 gcc 可以获得 C/C++语言头文件的系统目录，而 cygpath 工具却可以将目录格式从 Linux 格式转换成 Windows 格式，因为 pc-lint 所需要的路径格式必须是 Windows 格式的。如何获得 C/C++语言的系统头文件目录请参见 4.3.3 节，而图 30.4 示例说明了如何使用 cygpath 工具将一个 Linux 格式的路径转换为 Windows 格式。

```
/embedded/tools/lint
$ cygpath -w -p /usr/lib/gcc/i686-pc-cygwin/4.3.4/include/c++
C:\cygwin\lib\gcc\i686-pc-cygwin\4.3.4\include\c++
```

图 30.4

在 options.lnt 文件中，除了第 13～18 行用于指示所需系统头文件的目录外，还存在大量的以-e 开头加上一些数字的行。在 pc-lint 中每一个错误或警告都是通过一个数字来表示的，每一个数字所表达的具体含义可以从 pc-lint 的参考手册中查到。在 options.lnt 文件中，通过-e 加上一个数字的目的在于让 pc-lint 不需要报告数字所代表的问题。

第 13～18 行所指示的目录将被 pc-lint 当做是系统目录，而系统目录中的语义问题我们并不关心，为此需要采用-wlib（0）（options.lnt 中的第 20 行）选项告诉 pc-lint，"请不要告诉我任何与系统头文件有关的错误和警告"。抑制这类问题将为我们节省大量的时间，毕竟没有人希望在进行静态检查时，还花大量的时间查看系统文件中的问题。别忘了，对于我们来说项目代码才是静态分析的主体对象。

要了解 pc-lint 比编译器进行更为严格的语义检查，读者可以从图 30.5 的示例程序体会到。图中是一段将 value 的值转换成 2 的 *n* 次方的小程序，当采用 gcc 编译器进行编译时从图 30.6 中可以看出，它没有报告任何的警告；而采用 pc-lint 进行检查时，则报告第 12 行的程序存在一个 Info 级别的信息，其大意是程序对一个有符号的整型数进行了移位操作。我们知道，对一个有符号的整形数进行左移操作有可能改变值的正、负性。

```
00001: #include <stdio.h>
00002:
00003: int main ()
00004: {
00005:     int value = 0x10, mask_bit = 0x01;
```

```
00006:     int bits = 0, width = 32;
00007:
00008:     while (width > 0) {
00009:         if (0 != (value & mask_bit)) {
00010:             break;
00011:         }
00012:         mask_bit <<= 1;
00013:         bits ++;
00014:     }
00015:
00016:     printf ("value = %x, bits = %d\n", value, bits);
00017:     return 0;
00018: }
```

图 30.5

```
                    /embedded/tools/lint
$ gcc -Wall main.c -o main.exe

                    /embedded/tools/lint
$ ./lint-nt.exe -v std.lnt main.c
PC-lint for C/C++ (NT) Vers. 8.00x, Copyright Gimpel Software 1985-2008

      mask_bit <<= 1;
main.c  12  Info 701: Shift left of signed quantity (int)
```

图 30.6

要去除这个 Info，需要对代码进行修改，如图 30.7 的第 5 行，将 value 和 mask_bit 变量都定义成无符号的整型就行了。

```
00001: #include <stdio.h>
00002:
00003: int main ()
00004: {
00005:     unsigned int value = 0x10, mask_bit = 0x01;
00006:     int bits = 0, width = 32;
00007:
00008:     while (width > 0) {
00009:         if (0 != (value & mask_bit)) {
00010:             break;
00011:         }
00012:         mask_bit <<= 1;
00013:         bits ++;
00014:     }
00015:
00016:     printf ("value = %x, bits = %d\n", value, bits);
00017:     return 0;
00018: }
```

图 30.7

图 30.8 是另一个例子，图 30.9 是对它采用编译器及 pc-lint 进行检查的结果。相似地，编译器没有报告任何的警告，但 pc-lint 却报告了一个警告和一个 Info 级别的信息。因为 pc-lint 发现了在代码的第 16 行和第 17 行之间少了一个 break 语句。

```
00001: #include <stdio.h>
00002:
00003: typedef enum {
00004:     STATE_WAIT,
00005:     STATE_PROCESS,
00006:     STATE_INVALID
00007: } state_t;
00008:
00009: int main ()
00010: {
00011:     state_t state = STATE_WAIT;
00012:
00013:     switch (state)
00014:     {
00015:     case STATE_WAIT:
00016:         printf ("state is STATE_WAIT!\n");
00017:     case STATE_PROCESS:
00018:         printf ("state is STATE_PROCESS!\n");
00019:         break;
00020:     case STATE_INVALID:
00021:         printf ("state is STATE_INVALID!\n");
00022:         break;
00023:     }
00024:
00025:     return 0;
00026: }
```

图 30.8

```
                    /embedded/tools/lint
$ gcc -Wall main.c -o main.exe

                    /embedded/tools/lint
$ ./lint-nt.exe -v std.lnt main.c
PC-lint for C/C++ (NT) Vers. 8.00x, Copyright Gimpel Software 1985-2008

    case STATE_PROCESS:
main.c  17  Warning 616: control flows into case/default
main.c  17  Info 825: control flows into case/default without -fallthrough comment
```

图 30.9

从程序的上下文来看，pc-lint 是对的，在第 16 行和第 17 行之间确实需要一个 break 语句。但在某些情形下，如果我们的确不需要一个 break 语句，但也不希望 pc-lint 报告这是一个问题，那该如何处理呢？图 30.10 示例说明了如何通过增加一行注释的方式以屏蔽 pc-lint 报告特定的信息。

```
00001: #include <stdio.h>
00002:
00003: typedef enum {
00004:     STATE_WAIT,
00005:     STATE_PROCESS,
00006:     STATE_INVALID
00007: } state_t;
00008:
00009: int main ()
00010: {
```

```
00011:     state_t state = STATE_WAIT;
00012:
00013:     switch (state)
00014:     {
00015:     case STATE_WAIT:
00016:        printf ("state is STATE_WAIT!\n");
00017:        //lint -fallthrough
00018:     case STATE_PROCESS:
00019:        printf ("state is STATE_PROCESS!\n");
00020:        break;
00021:     case STATE_INVALID:
00022:        printf ("state is STATE_INVALID!\n");
00023:        break;
00024:     }
00025:
00026:     return 0;
00027: }
```

图 30.10

从图中第 17 行可以看出，要屏蔽 pc-lint 所报告的一个信息，可以通过增加一行注释的方式，且注释必须是以“//lint”开头的。对于这样的注释行，pc-lint 看到了以后就会从中找到命令选项，进行信息抑制。那我们如何知道需要在“//lint”之后放什么样的命令选项呢？这需要查看 pc-lint 的参考手册，因为其中对于每一个 Info、Warning 或者 Error 都说明了如何进行信息抑制，以及应使用怎样的选项。

pc-lint-msg.txt 中的内容可以以 pc-lint 所报告的信息编号作为关键字进行查找，比如图 30.9 中报告了 616 和 825 两个信息号，在 pc-lint-msg.txt 中就可以找到采用-fallthrough 进行信息抑制。

除了采用-fallthough 选项外，还有一种通用的方法可以采用，那就是使用-e 选项，如图 30.11 所示。这种方式就是将 pc-lint 所报告的错误号放在-e 参数的后面，就能起到抑制对应信息的作用。

```
00001: #include <stdio.h>
00002:
00003: typedef enum {
00004:     STATE_WAIT,
00005:     STATE_PROCESS,
00006:     STATE_INVALID
00007: } state_t;
00008:
00009: int main ()
00010: {
00011:     state_t state = STATE_WAIT;
00012:
00013:     switch (state)
00014:     {
00015:     case STATE_WAIT:
00016:        printf ("state is STATE_WAIT!\n");
00017:        //lint -e616 -e825
00018:     case STATE_PROCESS:
00019:        printf ("state is STATE_PROCESS!\n");
00020:        break;
```

```
00021:     case STATE_INVALID:
00022:         printf ("state is STATE_INVALID!\n");
00023:         break;
00024:     }
00025:
00026:     return 0;
00027: }
```

图 30.11

请注意-e 选项抑制的是所在行的后一行代码。比如，图 30.11 的第 17 行所抑制的只能是它后面的一行，也就是少了的 break 语句。如果想在一个文件中抑制所有的同一问题，则可以在文件的开头使用“//lint -e{616, 825}”这样的形式。使用这一方法需要小心，因为它所抑制的面太大了，会造成抑制一些可能是问题的点，这会降低静态分析的效果。

图 30.12 示例说明了抑制针对 g_statistics 变量所报告的编号为 728 的错误，两种抑制方法中上一种更好，因为它只针对 g_statistics 变量；而下一种方法将针对同一文件中的所有变量，它的抑制面更广。

```
00048: //lint -esym(728, g_statistics)
00049: static device_statistics_t g_statistics;
```

```
00048: //lint -e{728}
00049: static device_statistics_t g_statistics;
```

图 30.12

30.2　无缝整合静态分析

从使用的角度来看，如果项目运行“make scheck”就能方便地对所有的源代码进行静态分析，那就太好不过了。目标名 scheck 是“static check”的简写形式。要将静态分析的功能嵌入到开发环境中，仍可以借助 make 工具做到。

pc-lint 工具有一个特点，就是当分析一个源文件时，如果出现任何的 Info、Warning 或者 Error 都将返回非 0 值，而这一特点正是 make 所希望的。

30.2.1　更改 Makefile

要将 pc-lint 嵌入到开发环境中，离不开更改 c.rule。由于 pc-lint 不仅能对 C 程序进行静态分析，还能对 C++程序进行静态分析，因此 c++.rule 也不可避免地需要被更改，但两者的更改如出一辙。在这里只对 c.rule 的更改进行介绍，更改后的 c.rule 文件如图 30.13 所示。

```
00001: MKDIR = mkdir
00002: RM = rm
00003: CHKER = lint-nt.exe
```

```
00004:
00005: RMFLAGS = -fr
00006: ARFLAGS = crs
00007: CFLAGS += -Wall -std=gnu99
00008: CHKFLAGS = -v std.lnt
00009:
00010: ......
00038:
00039: ifeq ("$(MAKECMDGOALS)", "scheck")
00040: DIR_OBJS = sobjs
00041: endif
00042:
00043: DIR_TARGET = $(BUILD)/$(MAKECMDGOALS)
00044: DIRS = $(DIR_OBJS) $(DIR_TARGET)
00045: DIR_CHKER = $(ROOT)/tools/lint
00046:
00047: SRCS = $(wildcard *.c)
00048: ASMS = $(wildcard *.S)
00049: UTS = $(wildcard unitest_*.c)
00050: OBJS := $(addprefix $(DIR_OBJS)/, $(SRCS:.c=.o))
                $(addprefix $(DIR_OBJS)/, $(ASMS:.S=.o))
00051: DEPS := $(addprefix $(DIR_OBJS)/, $(SRCS:.c=.dep))
                $(addprefix $(DIR_OBJS)/, $(ASMS:.S=.dep))
00052: CHKS := $(addprefix $(DIR_OBJS)/, $(SRCS:.c=.chk))
00053: RMS = robjs dobjs uobjs sobjs
00054:
00055: ......
00074:
00075: ifneq ("$(INCLUDE_DIRS)", "")
00076: ifeq ($(MAKECMDGOALS), scheck)
00077: CHK_INCLUDE_DIRS += $(INCLUDE_DIRS)
00078: CHK_INCLUDE_DIRS := $(shell cygpath -w -p $(CHK_INCLUDE_DIRS))
00079: CHK_INCLUDE_DIRS := $(addprefix ", $(CHK_INCLUDE_DIRS))
00080: CHK_INCLUDE_DIRS := $(addsuffix ", $(CHK_INCLUDE_DIRS))
00081: CHK_INCLUDE_DIRS := $(addprefix -I, $(CHK_INCLUDE_DIRS))
00082: CHK_INCLUDE_DIRS := $(strip $(CHK_INCLUDE_DIRS))
00083: endif
00084: INCLUDE_DIRS := $(addprefix -I, $(INCLUDE_DIRS))
00085: INCLUDE_DIRS := $(strip $(INCLUDE_DIRS))
00086: endif
00087:
00088: ......
00096:
00097: .PHONY: release debug clean unitest scheck
00098: release debug unitest: $(EXE) $(UTS) $(LIB)
00099: clean:
00100:     -$(RM) $(RMFLAGS) $(RMS)
00101: scheck: $(CHKS)
00102:
00103: ......
00106:
00107: ifeq ($(MAKECMDGOALS), scheck)
00108: CHK_CHKER_DIR += $(DIR_CHKER)
00109: CHK_CHKER_DIR := $(shell cygpath -w -p $(CHK_CHKER_DIR))
00110: CHK_CHKER_DIR := $(addprefix ", $(CHK_CHKER_DIR))
00111: CHK_CHKER_DIR := $(addsuffix ", $(CHK_CHKER_DIR))
00112: CHK_CHKER_DIR := $(addprefix -I, $(CHK_CHKER_DIR))
00113: CHK_CHKER_DIR := $(strip $(CHK_CHKER_DIR))
00114: CHK_ROOT_DIR := $(shell cygpath -w -p $(ROOT))
00115: endif
00116:
```

```
00117: ......
00130: $(DIR_OBJS)/%.dep: $(DEP_DIR_OBJS) %.c
00131:     @echo "Creating $@ ..."
00132:     @set -e ; \
00133:     $(RM) $(RMFLAGS) $@.tmp ; \
00134:     $(CC) $(INCLUDE_DIRS) -E -MM $(filter %.c, $^) > $@.tmp ; \
00135:     sed 's,\(.*\)\.o[ :]*,$(DIR_OBJS)/\1.o
               $@ $(DIR_OBJS)/\1.chk: ,g' < $@.tmp > $@ ; \
00136:     $(RM) $(RMFLAGS) $@.tmp ; \
00137:     if [ -n "$(UTS)" ] ; then echo "$(DIR_TARGET)/$*.exe:
               $(DIR_OBJS)/$*.o" >> $@ ; fi
00138: ......
00145: $(DIR_OBJS)/%.chk: $(DEP_DIR_OBJS) %.c
00146:     $(DIR_CHKER)/$(CHKER) $(CHK_CHKER_DIR) $(CHK_INCLUDE_DIRS)
              $(CHKFLAGS) -libdir "($(CHK_ROOT_DIR)\*)" $(filter %.c, $^)
00147:     @touch $@
00148:
```

图 30.13

c.rule 中的更改包含如下几处。

（1）新增的第 3 行定义了 CHKER 变量用于记录 pc-lint 可执行程序的文件名。

（2）位于第 8 行新增的 CHKFLAGS 变量用于保存 pc-lint 所需要的（部分）命令参数。

（3）增加第 39～41 行的目的是，当运行“make scheck”时，将生成的文件放入到 sobjs 目录中。sobjs 是“static objects”的简写。实际上，使用 pc-lint 进行静态分析时并不会生成文件，它是将所有的信息直接输出到终端上。之所以要生成文件，是为了提高项目的静态分析效率。如果一个文件成功完成了静态分析，在没有任何更改的情形下，在下一次运行“make scheck”时我们希望跳过对它的分析以提高效率。这与编译器编译程序的道理是一样的。为了做到这一点，需要建立合适的依赖关系。当一个文件被成功分析完后，就在 sobjs 目录下生成一个对应的.chk 文件。通过在.c 与.chk 文件之间建立依赖关系，就可以做到在分析时跳过没有更改的文件。

（4）第 45 行的 DIR_CHKER 变量记录了 lint-nt.exe 所在目录的路径。

（5）新增的第 52 行用于获得“make scheck”所需创建的.chk 文件列表。注意，.chk 文件的生成就意味着所对应的.c 文件成功地通过了静态分析。

（6）第 53 行在 RMS 变量中增加了 sobjs 目录，因为我们希望运行“make clean”时 sobjs 目录也被删除。

（7）第 76～83 行用于处理头文件目录，且这一处理只会发生在被创建的目标是 scheck 时。INCLUDE_DIRS 中存放的是一个模块被编译时所需头文件所在的目录，在进行 pc-lint 静态分析时，这些目录中的头文件同样是必需的。但是，前面提到了，pc-lint 所需的目录路径形式是 Windows 格式的，因此需要通过调用 cygpath 工具将 Linux 格式转换成 Windows 格式，并通过 -I 选项将所需的目录引入到 pc-lint 中，这一点与 options.lnt 中对-I 选项的运用是相同的。第 79 和 80 行的作用是将 Windows 路径用引号括起来。第 81 行则在每一个被引号括起来的路径前加上一个-I。第 82 行则去除各路径之间的多余空格，以便它们被打印到终端上时更美观。

（8）第 97 行将 scheck 定义为一个假目标。

（9）第 101 行则增加了构建 scheck 目标的规则，且让 scheck 目标依赖于.chk 文件。

（10）第 107～115 行的作用与第 76～83 行是相似的，只不过处理的目录是 pc-lint 所在的目录。之所以需要将 pc-lint 所在的目录也通过-I 选项告知 pc-lint，是因为 std.lnt 存在于该目录中，如果不指定 pc-lint 将报告找不到该文件。注意，第 114 行是将整个项目的$(ROOT)路径转换为 Windows 格式。

（11）第 135 行为.chk 文件建立依赖关系，这一点与.o 文件的依赖关系是完全一样的。有了这一依赖关系后，make 就知道何时应当再一次调用 pc-lint 对文件进行静态分析并生成.chk 文件。

（12）第 145～147 行定义了.chk 文件的生成规则。在生成.chk 文件之前，需要先调用 pc-lint 对代码进行分析（第 146 行），如果分析通过了，则通过 touch 命令生成一个.chk 文件（第 147 行），以表示相应的文件成功地通过了静态分析。需要提醒一下，如果 pc-lint 发现被分析的文件有问题它会返回非 0 值，进而 make 将立即终止运行，而不会继续运行第 147 行的 touch 命令。注意，在调用 pc-lint 进行静态分析时，用到了-libdir 选项，这个选项与 options.lnt 中的-wlib（0）选项相匹配。前面曾指出，-wlib（0）的目的是抑制对系统文件进行静态分析，而系统文件在 options.lnt 中是通过-I 选项来告知 pc-lint 的。在 c.rule 中，也将项目目录以-I 的形式告诉了 pc-lint，pc-lint 显然不知道哪一个-I 选项所指定的是系统目录或项目目录，对于它来说全都是系统目录，结果就是 pc-lint 也不对项目目录进行静态分析，这显然不是我们所期望的。通过-libdir 选项可以告诉 pc-lint，对于所有位于 ROOT 目录下的文件都不要将其视为系统文件。

图 30.14 则以 dll.c 文件为例，示例说明了 c.rule 中的改动在依赖关系中的位置。

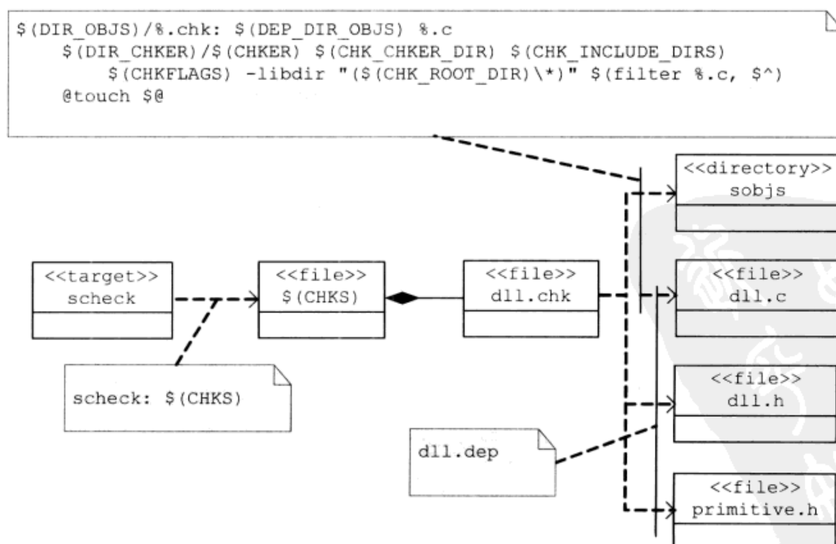


图 30.14

图 30.15 示例说明了在新的 c.rule 下，dll.c 的依赖关系文件 dll.dep 的内容，请注意其中新增了 sobjs/dll.chk 这一目标。

```
/embedded/buildv4
$ cat ../code/platform/common/src/sobjs/dll.dep
sobjs/dll.o sobjs/dll.dep sobjs/dll.chk: dll.c \
/embedded/code/platform/common/inc/dll.h \
    /embedded/code/platform/common/inc/primitive.h
```

图 30.15

除了对 c.rule 和 c++.rule 文件的更改外，embedded/build/Makefile 文件也需要更改，其改动如图 30.16 所示。改动包含：

（1）新增的第 75～77 行是将 NONUT_DIRS 变量中的目录加入到 MAKE_DIRS 变量中，表示其中的文件也需要进行静态分析。

（2）第 82 行将 scheck 目录放入 RMS 变量中，以便在“make clean”时将它删除。当运行“make scheck”时，scheck 目录仍会在 embedded/buildv4 目录下被创建，尽管其中什么文件都不放。这样做是从一致性角度出发的。

（3）第 87 行将 scheck 定义为假目标。

（4）第 88 行在规则中增加了 scheck 目标。

```
00074: ......
00075: ifeq ("$(MAKECMDGOALS)", "scheck")
00076: MAKE_DIRS += $(NONUT_DIRS)
00077: endif
00078:
00079: RM = rm
00080: MKDIR = mkdir
00081: RMFLAGS = -fr
00082: RMS = release debug unitest $(DIR_COVERAGE) scheck
00083:
00084: DIR_UNITEST = unitest
00085: DIR_COVERAGE = coverage
00086:
00087: .PHONY: release debug clean touch unitest test force creport scheck
00088: release debug clean unitest scheck:
00089:     @set -e; \
00090:     for DIR in $(MAKE_DIRS); \
00091:         do \
00092:             cd $$DIR && $(MAKE) -r ROOT=$(ROOT) BUILD=$(BUILD) $@;\
00093:         done
00094:     @set -e; \
00095:     if [ "$(MAKECMDGOALS)" = "clean" ] ;
then $(RM) $(RMFLAGS) $(RMS) ; fi; \
00096:     if [ "$(MAKECMDGOALS)" = "unitest" ] ;
then touch $(DIR_UNITEST)/unitested; fi
00097:     @echo ""
00098:     @echo ":-) Completed"
00099:     @echo ""
00100: ......
```

图 30.16

对于 embedded 项目，还有另一个 Makefile 文件需要更改，那就是用于生成 err2str 小程序的 Makefile，图 30.17 示例说明了其中的更改。

```
00001: EXE = err2str.exe
00002: LIB =
00003:
00004: INCLUDE_DIRS =
00005:
00006: LINK_LIBS =
00007:
00008: include $(BUILD)/c++.rule
00009:
00010: scheck: release
00011:
```

图 30.17

在第 15 章介绍了如何通过 err2str 小程序来辅助 errstr()函数的实现，在静态分析时我们仍需要 err2str 小程序帮助生成相应的文件，否则 pc-lint 在对相应的文件进行静态分析时会因为找不到文件而报错。为此，我们需要在图 30.17 中增加其中的第 10 行，即在构建 scheck 时，先构建 release 目标，也就是将生成 err2str 小程序。为了理解这一点，读者可以通过注释掉第 10 行以查看将出现怎样的错误。

30.2.2 检查整合效果

对 Makefile 更改以后，在 embedded/buildv4 目录下运行“make scheck”，将会看到 pc-lint 被运用于对每一个源文件进行静态分析。当对整个项目都完成了分析后，在终端上最终会看到“:-) Completed”，如图 30.18 所示。

```
                    /embedded/buildv4
$ make scheck
make[1]: Entering directory `/embedded/code/tools/err2str'
/embedded/buildv4/c++.rule:103: sobjs/err2str.dep: No such file or directory
mkdir sobjs
Creating sobjs/err2str.dep ...
make[1]: Leaving directory `/embedded/code/tools/err2str'
make[1]: Entering directory `/embedded/code/tools/err2str'
/embedded/tools/lint/lint-nt.exe -I"C:\cygwin\embedded\tools\lint"
-v std.lnt -libdir"(C:\cygwin\embedded\*)" err2str.cpp
PC-lint for C/C++ (NT) Vers. 8.00x, Copyright Gimpel Software 1985-2008
...... 显示结果有删减 ......
mkdir sobjs
Creating sobjs/main.dep ...
make[1]: Leaving directory `/embedded/code/application/sync/recursive'
make[1]: Entering directory `/embedded/code/application/sync/recursive'
/embedded/tools/lint/lint-nt.exe -I"C:\cygwin\embedded\tools\lint"
-I"C:\cygwin\embedded\code\platform\common\inc"
-I"C:\cygwin\embedded\code\platform\arch\x86\simulator\inc"
-I"C:\cygwin\embedded\code\platform\device\inc"
-I"C:\cygwin\embedded\code\platform\timer\v3\inc"
-I"C:\cygwin\embedded\code\platform\task\common\inc"
-I"C:\cygwin\embedded\code\platform\task\v3\inc"
```

```
-I"C:\cygwin\embedded\code\platform\sync\v3\inc" -v std.lnt
-libdir"(C:\cygwin\embedded\*)" main.c
PC-lint for C/C++ (NT) Vers. 8.00x, Copyright Gimpel Software 1985-2008
make[1]: Leaving directory `/embedded/code/application/sync/recursive'

:-) Completed
```

图 30.18

当成功地完成了对项目的静态分析以后，如果在没有更改项目源程序的情况下再一次运行“make scheck”，将发现 pc-lint 并不会真正被用于进行静态分析。这种效果正是因为我们通过.chk 文件建立了完备的依赖关系树，以至 make 知道在这种情形下不需要调用 pc-lint 对任何源文件进行静态分析。很显然，这样的静态分析环境将显著地提高工作效率。

30.3 小结

由于 C 编程语言本身存在的不严谨性，一个文件即使被成功编译了，但仍有可能因为存在的语义错误，使得交付给用户的软件中存在潜在的缺陷。通过使用静态分析工具对项目进行更为细致的语义分析，有助于减少软件缺陷。

一个工具要被使用，我们就考虑其易用性。通过将静态分析工具无缝地整合到开发环境中将显著地提高它的易用性。

第 31 章

动态分析，使程序更健壮

上一章所介绍的 pc-lint 静态分析工具除了能发现程序语义上的（潜在）错误外，还能发现一些具有局部性的相对复杂的问题。图 31.1 和图 31.2 中两个小程序中的错误，pc-lint 都能发现。

```
00001: int main ()
00002: {
00003:     char a [10];
00004:     a [-1] = 10;
00005:     return 0;
00006: }
```

图 31.1

```
00001: int main ()
00002: {
00003:     void *p = malloc (sizeof (char));
00004:     return 0;
00005: }
```

图 31.2

图 31.1 中的第 4 行存在内存写溢出问题，而图 31.2 中第 3 行存在所分配的内存忘记释放问题。pc-lint 之所以能发现错误，是因为它们具有很强的局部性，即错误完全可以根据单个函数中的内容进行界定。如果将图 31.1 的实现稍做改动，即得到图 31.3 所示的实现，pc-lint 就无能为力了。我们知道，图 31.3 中的错误与图 31.1 是一样的，而要检查出这样的问题必须通过动态分析才能做到。动态分析工具可用于发现内存非法访问、内存泄露等问题。

```
00001: void foo (int _index)
00002: {
00003:     char a [10];
00004:     a [_index] = 10;
00005: }
00006:
00007: int main ()
00008: {
00009:     foo (-1);
00010:     return 0;
00011: }
```

图 31.3

与静态分析工具不同的是，动态分析工具必须通过运行被检查代码所编译生成的程序来完成，这应当是“动态”一词的来源。大部分动态分析工具在动态分析之前，要求使用它所提供的编译工具重新编译程序，以便“偷偷”地插入一些代码，从而实现动态分析。

代码动态分析工具有多种选择，比如商用软件有 IBM 的 Purify、Parasoft 的 Insure++等，以及目前来自开源世界并被广泛采用的 valgrind。本章将以 valgrind 为例，介绍如何将它无缝地整合到开发环境中。选择 valgrind 工具很重要的一个原因是它是免费的，与其他价格不菲的商用软件相比这是很大的一个优势。另一个原因就是其易用性，使用 gcc/g++编译出来的程序可以直接用 valgrind 进行分析，而无须像 Purify 和 Insure++那样要重新编译。

31.1 结识动态分析工具

valgrind 只能运行在 Linux 操作系统上。由此看来，如果一个模块希望采用 valgrind 工具进行动态分析，则必须让这一模块能在 Linux 操作系统上运行。如果读者的嵌入式操作系统刚好不是 Linux，就需要运用跨平台技术将软件模块设计成能运行于 Linux 操作系统中。

如果读者有现成的 Linux 主机，则可以在主机上运行 valgrind 命令以检查它是否存在，如图 31.4 所示。

```
valgrind --version
valgrind-3.5.0
```

图 31.4

如果不存在，则需要从 valgrind 的官网 www.valgrind.org 下载并在 Linux 主机上进行编译和安装。对于使用本书光盘中所带虚拟机的读者，valgrind 在虚拟机中已经安装了。图 31.5 以从 valgrind 官网上下载的 valgrind-3.5.0.tar.bz2 源码包为例，示例说明了编译和安装 valgrind 的所有关键步骤。

```
pwd
/usr/src
ls
valgrind-3.5.0.tar.bz2
bunzip2 valgrind-3.5.0.tar.bz2
tar xf valgrind-3.5.0.tar
cd valgrind-3.5.0/
./configure
make
make install
```

图 31.5

为了查看 valgrind 的功效，让我们以图 31.6 中的小程序为例，该程序的功能是读取文件并将文件内容显示在终端上。

```
main.c
00001: #include <stdio.h>
00002: #include <stdlib.h>
00003:
00004: int main (int _argc, const char *_argv [])
00005: {
00006:    FILE *fp;
00007:    char *line = 0;
00008:    size_t len = 0;
00009:    ssize_t read;
00010:
00011:    if (_argc != 2) {
00012:       printf ("Invalid parameter!\n");
00013:       return -1;
00014:    }
00015:
00016:    fp = fopen (_argv[1], "r");
00017:    if (0 == fp) {
00018:       printf ("Cannot open file: %s\n", _argv [1]);
00019:       return -1;
00020:    }
00021:
00022:    while ((read = getline (&line, &len, fp)) != -1) {
00023:       printf ("%s", line);
00024:       line = 0;
00025:    }
00026:
00027:    if (line) {
00028:       free (line);
00029:    }
00030:    fclose (fp);
00031:
00032:    return 0;
00033: }
```

图 31.6

图 31.7 示例说明了如何对这个小程序进行编译并检查其运行效果。

```
gcc main.c -o main.exe
./main.exe main.c
#include <stdio.h>
#include <stdlib.h>

int main (int _argc, const char *_argv [])
{
   FILE *fp;
   char *line = 0;
   size_t len = 0;
   ssize_t read;

   if (_argc != 2) {
      printf ("Invalid parameter!\n");
      return -1;
   }

   fp = fopen (_argv[1], "r");
   if (0 == fp) {
      printf ("Cannot open file: %s\n", _argv [1]);
      return -1;
   }
```

```
    while ((read = getline (&line, &len, fp)) != -1) {
        printf (" s", line);
        line = 0;
    }

    if (line) {
        free (l_ne);
    }
    fclose (pf);

    return 0;
}
```

图 31.7

这个示例小程序在功能上并不存在问题，在图 31.7 所示的测试结果中，它正确地将 main.c 中的内容显示出来，但它却存在内存泄漏问题。读者注意到了吗？或许回头重新检查程序仍发现不了其中的内存泄漏点，那让 valgrind 来显显身手吧。图 31.8 示例说明了运用 valgrind 进行动态分析的结果。

```
                              ~ valgrind --leak-check=full --show-reachable=yes ./main.exe
    main<.c
...... 显示结果有删减 ......
==30354== HEAP SUMMARY:
==30354==     in use at exit: 3,960 bytes in 33 blocks
==30354==   total heap usage: 35 allocs, 2 frees, 4,432 bytes allocated
==30354==
==30354== 3,960 bytes in 33 blocks are definitely lost
==30354==    at 0x40058E3: malloc (vg replace malloc.c:195)
==30354==    by 0xC2D9A2: getdelim (in /lib/libc-2.5.so)
==30354==    by 0xC2ADD2: getline (in /lib/libc-2.5.so)
==30354==    by 0x8048520: main (in /root/main.exe)
==30354==
==30354== LEAK SUMMARY:
==30354==     definitely lost: 3,960 bytes in 33 blocks
==30354==     indirectly lost: 0 bytes in 0 blocks
==30354==       possibly lost: 0 bytes in 0 blocks
==30354==     still reachable: 0 bytes in 0 blocks
==30354==          suppressed: 0 bytes in 0 blocks
==30354==
==30354== For counts of detected and suppressed errors, rerun with: -v
==30354== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 13 from 8)
```

图 31.8

使用 valgrind 进行动态分析，只需运行它并将要检查的可执行程序作为它的参数就行了，并不存在需要对程序进行重新编译的麻烦。在 valgrind 最后输出的报告中，指出被检查程序中存在一处内存泄漏问题。泄漏是由 main()函数所调用的 getline()引起的，最终造成了 3960 字节的泄漏。

内存泄漏是因为其中多余的第 24 行。通过 getline()函数的参考手册我们将发现，getline()会为应用程序分配内存用于存取返回行的内容，而当调用 getline()函数所指定的第一个参数不为 NULL 时，它将释放它所指向的内存，而多余的第 24 行每次都将 line 变量置为 NULL，从而造成 getline()函数认为不需要进行内存释放操作。读者可以去除其中的第 24 行，编译后再一

次运行 valgrind 对程序进行检查。

通过图 31.6 所示的这个小程序示例，相信读者明白了对程序进行动态分析的重要性和必要性，也看到了 valgrind 工具是如何被使用的。接下来看一看如何将 valgrind 整合到开发环境中。

31.2　无缝整合动态分析

进行动态分析同样需要三个步骤。第一步是生成包含被检查代码的可执行程序；第二步是通过 valgrind 运行生成的可执行程序进行动态分析；第三步是获得检查报告以了解结果。

在 30.2 节中引入了 scheck 目标以对项目进行静态分析这一功能，那增加 dcheck 目标表示动态分析就显得很自然了。dcheck 是“dynamic check”的简写。另外，在第 29 章通过将代码覆盖设计成以单元测试为中心，对动态分析也可以采用这种形式。因为运行“make unitest”将生成用于单元测试的可执行程序，这一可执行程序刚好可以通过使用 valgrind 进行动态分析。

31.2.1　更改 Makefile

在 Makefile 中创建 dcheck 目标与创建 scheck 目标所需的更改将完全不同。在创建 scheck 目标时，由于静态分析是针对每一个源程序文件进行的，因此不可避免地需要对 c.rule 和 c++.rule 两个文件进行更改。与之不同的是，运用 valgrind 进行动态分析并不是基于源程序文件的，而是针对最终编译出来的单元测试可执行程序的，从这一点来说，dcheck 目标的实现与 test 目标更像。

进行动态分析需要两个步骤。第一步是借助 valgrind 运行可执行程序，这可以通过“make dcheck”做到；第二步则是检查动态分析结果，这需要通过运行“make dreport”来实现。dreport 是“dynamic report”的简写。图 31.9 示例说明了对 embedded/buildv5/Makefile 文件的更改。

```
00086: ......
00087: .PHONY: release debug clean touch unitest test force creport scheck dcheck dreport
00088: ......
00103: ifeq ("$(MAKECMDGOALS)", "test")
00104: ifeq ("$(wildcard $(DIR_UNITEST)/unitested)", "")
00105: $(error Did you forget to run 'make unitest'?)
00106: endif
00107:
00108: $(DIR_UNITEST)/unitest_%: force
00109:     ./$@
00110: endif
00111:
00112: $(DIR_UNITEST)/unitest_%: force
00113:     ./$@
00114:
00115: test: $(UTS)
00116: @touch $(DIR_UNITEST)/tested
00117:
00118: force:
00119: ......
00156: ifeq ("$(MAKECMDGOALS)", "dcheck")
```

```
00157: ifeq ("$(wildcard $(DIR_UNITEST)/unitested)", "")
00158: $(error Did you forget to run 'make unitest'?)
00159: endif
00160:
00161: $(DIR_UNITEST)/unitest_%: force
00162:     valgrind --leak-check=full --track-origins=yes --read-var-info=yes
00162:         --malloc-fill=0xFF --log-file=$@.mem.log ./$@
00163:     valgrind --tool=exp-ptrcheck --enable-sg-checks=yes --log-file=$@.ptr.log ./$@
00164: endif
00165:
00166: dcheck: $(UTS)
00167:     @touch $(DIR_UNITEST)/dchecked
00168:     @echo ""
00169:     @echo ":-) Completed"
00170:     @echo ""
00171:
00172: ifeq ("$(MAKECMDGOALS)", "dreport")
00173: ifeq ("$(wildcard $(DIR_UNITEST)/dchecked)", "")
00174: $(error Did you forget to run 'make unitest' and 'make dcheck'?)
00175: endif
00176: endif
00177:
00178: dreport:
00179:     @cd $(DIR_UNITEST) && grep "ERROR SUMMARY" *.log |
            sed 's,\(.*\)\.log\(.*\)ERROR SUMMARY[ :]\(.*\),\1.log summary: \n    \3,g'
00180:     @echo ""
00181:     @echo ":-) Generated"
00182:     @echo ""
```

图 31.9

Makefile 文件中的更改包含如下几处。

（1）第 87 行将 dcheck 和 dreport 定义为假目标。

（2）第 112～113 行被移到了第 108～109 行，即只有在构建 test 目标时才让这一规则发挥作用。因为我们还得在构建 dcheck 目标时建立类似的一个规则。

（3）第 157～159 行用于判断在运行“make dcheck”之前，是否已运行过“make unitest”。这是因为在进行动态分析时，需要所有的单元测试可执行程序已被构建出来。

（4）第 161～163 行的规则与第 108～109 行的很相似，只不过是通过 valgrind 工具来运行单元测试可执行程序，这一规则只有在构建 dcheck 目标时才起作用。使用 valgrind 进行动态分析时，分析结果将被放入日志文件中。第 162 行的功能是使用 valgrind 进行内存泄漏检测，而第 163 行的功能是使用 valgrind 对非法指针进行检查。

（5）第 166～170 行定义了构建 dcheck 目标的规则。dcheck 目标依赖于$(UTS)，这一依赖关系将造成第 161～163 行的规则会针对每一个单元测试可执行程序被 valgrind 运行以进行动态分析。第 167 行是在对所有的可执行程序完成了动态分析之后，在 embedded/buildv5/unitest 目录下创建一个 dchecked 文件，以表示对项目完成了动态分析。这一文件在构建 dreport 目标时将用到。

（6）第 172～176 行用于判断当构建 dreport 目标时，是否已运行过“make dcheck”，这是

通过检查 embedded/buildv5/unitest 目录下的 dchecked 文件来实现的。

（7）第 178～182 行定义了构建 dreport 的规则。获取动态分析报告的实现很简单，只是通过 grep 工具将所有的日志文件中的“ERROR SUMMARY”提取出来。其中如果出现“ERROR SUMMARY”的结果不为 0 时，则说明存在错误。

31.2.2　检查整合效果

检查效果可以通过运行“make unitest”、“make dcheck”和“make dreport”三个步骤。图 31.10 所示是运行完“make dreport”后所获得的分析报告。

```
                        /embedded/buildv5  make dreport
unitest bitmap.exe.mem.log summary:
   0 errors from 0 contexts (suppressed: 13 from 8)
unitest bitmap.exe.ptr.log summary:
   0 errors from 0 contexts (suppressed: 0 from 0)
unitest clib.exe.mem.log summary:
   0 errors from 0 contexts (suppressed: 13 from 8)
unitest clib.exe.ptr.log summary:
   0 errors from 0 contexts (suppressed: 0 from 0)
unitest dll.exe.mem.log summary:
   0 errors from 0 contexts (suppressed: 13 from 8)
unitest dll.exe.ptr.log summary:
   0 errors from 0 contexts (suppressed: 0 from 0)
unitest dllht.exe.mem.log summary:
   0 errors from 0 contexts (suppressed: 13 from 8)
unitest dllht.exe.ptr.log summary:
   0 errors from 0 contexts (suppressed: 0 from 0)
unitest task.exe.mem.log summary:
   533634 errors from 715 contexts (suppressed: 13 from 8)
unitest task.exe.ptr.log summary:
   2 errors from 2 contexts (suppressed: 0 from 0)

:-) Generated
```

图 31.10

从图中的显示结果来看，unitest_task.exe 文件中存在问题，它所对应的被测试文件为 task.c。读者可以通过浏览 unitest_task.exe.mem.log 和 unitest_task_exe. ptr.log 文件来了解 valgrind 所报告的错误。经过作者检查，这些并非真正的错误，可能是因为任务管理模块中的情景切换而引起 valgrind 的误判。除了 unitest_task.exe 外，其他的被检查模块中都没有发现问题。

31.3　小结

动态分析能有效地发现程序中存在的非正常指针使用和内存泄漏等问题，这些错误通过静态分析是无法发现的。通过实施动态分析有助于提高程序的健壮性。

本章示例说明了如何将 valgrind 这一动态分析工具无缝地整合到开发环境中。

第 32 章 性能分析，让优化程序有的放矢

性能分析是通过使用工具分析程序中各函数的执行效率，以便找到性能瓶颈进行优化。程序的执行效率来源于合理的数据结构设计，以及正确地使用编程语言的特性和库函数。在理想情况下，我们应当养成基于模块检查程序性能的习惯，而不是等性能问题出现以后才开始检查。这是一种防患于未然的方式。

32.1 初探性能分析工具

上一章所介绍的 valgrind 其实是一个工具集，在它的集合中，有一个用于分析程序性能的工具——callgrind。除了使用 callgrind 外，还需要使用另一个开源项目——kcachegrind，它是一个具有图形界面的工具，以图形化的形式显示使用 callgrind 所获得的性能分析数据。在大多的系统中，kcachegrind 需要我们自己安装，读者可以通过搜索 Internet 了解如何安装它。在光盘内的虚拟机中该工具已经安装好了。

图 32.1 的示例程序用于帮助我们了解性能分析工具的用途。这个程序非常简单，其中实现了 foo()和 bar()函数，两个函数的实现是相似的，只不过累加计数的范围有所不同。

```
void foo()
{
   int sum = 0, i;

   for (i = 1; i < 10; i ++) {
      sum += i;
   }
}

void bar()
{
   int sum = 0, i;

   for (i = 1; i < 20; i ++) {
      sum += i;
   }
}
```

```
int main()
{
    foo();
    bar();
    return 0;
}
```

图 32.1

使用图 32.2 所示的命令编译 profile.c 文件。注意，需要使用-g 选项。接着运行 valgrind，以获取性能分析数据文件 profile.prof。

```
gcc -g profile.c -o profile.exe
valgrind -q --tool=callgrind --collect-jumps=yes
  --callgrind-out-file=profile.prof ./profile.exe
```

图 32.2

通过运行图 32.3 中的命令或像图 32.4 那样通过菜单启动 kcachegrind 程序。

```
/embedded/build  kcachegrind &
```

图 32.3

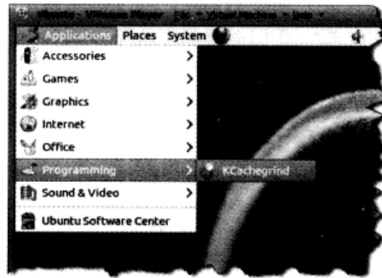


图 32.4

通过 kcachegrind 的[File]->[Open]菜单可以获得图 32.5 所示的对话框，用于选择需要打开的性能分析文件 profile.prof。注意，在对话框中需要将 Filter 选择为“All Files”，否则看不到以.prof 为后缀的性能分析数据文件。

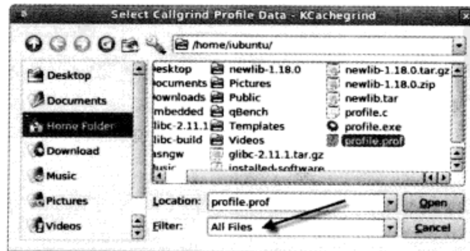


图 32.5

图 32.6 显示了 profile.c 文件中各函数在程序运行时所占用处理器时间的百分比。由于 bar()函数累加的范围更大，所以所占用的处理器时间比 foo()函数更多。除了查看各函数所耗时间占

整个程序的百分比外，还可以查看被调用函数占调用函数的百分比，如图 32.7 所示。从图中可以看出，foo()函数所花的时间占 main()函数的 32.93%。

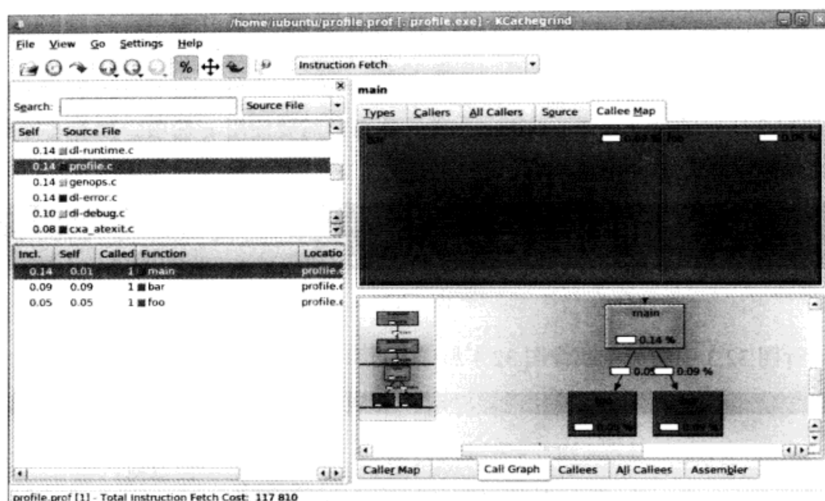


图 32.6

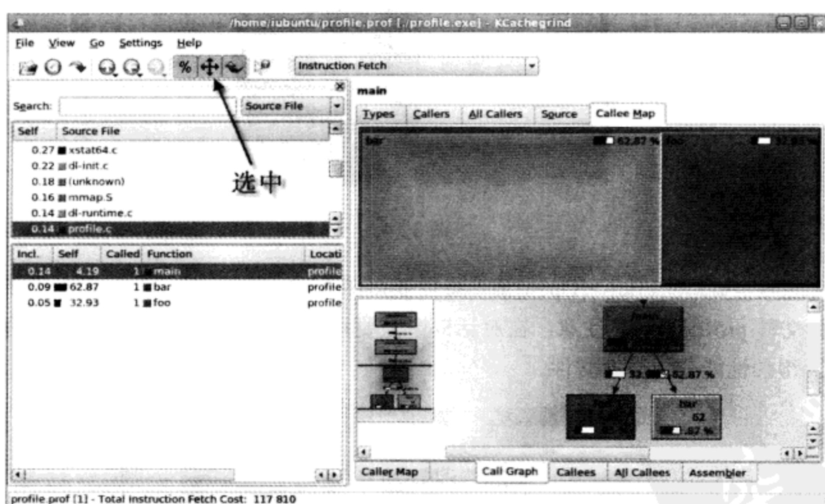


图 32.7

在 kcachegrind 内，还有其他的一些功能，读者可以自行摸索，在此不打算展开介绍。

32.2　无缝整合性能分析

性能分析同样需要运行被检查代码所编译生成的可执行程序，我们同样可以采用以单元测

试为中心的方式，方便地实现性能分析。当在 Makefile 中增加了 profile 目标后，通过“make unitest”和“make profile”两步将获得各单元测试可执行程序的性能分析数据。

32.2.1　更改 Makefile

将性能分析整合到开发环境只需更改 Makefile 文件。图 32.8 所示是更改后的 Makefile。

```
embedded/build/Makefile
00078: ......
00079: RM = rm
00080: MKDIR = mkdir
00081: MV = mv
00082: RMFLAGS = -fr
00083: MKDIRFLAGS = -p
00084: RMS = release debug unitest $(DIR_COVERAGE) scheck $(DIR_PROFILE)
00085:
00086: DIR_UNITEST = unitest
00087: DIR_COVERAGE = coverage
00088: DIR_PROFILE = profile
00089:
00090: .PHONY: release debug clean touch unitest test force creport scheck dcheck
        dreport profile
00091: ......
00185: ifeq ("$(MAKECMDGOALS)", "profile")
00186: ifeq ("$(wildcard $(DIR_UNITEST)/unitested)", "")
00187: $(error Did you forget to run 'make unitest'?)
00188: endif
00189:
00190: $(DIR_UNITEST)/unitest_%: force
00191:     valgrind -q --tool=callgrind --collect-jumps=yes
            --callgrind-out-file=$(basename $@).prof ./$@
00192: endif
00193:
00194: profile: $(UTS)
00195:     @$(MKDIR) $(MKDIRFLAGS) $(DIR_PROFILE) && $(MV) $(DIR_UNITEST)/*.prof
            $(DIR_PROFILE)
00196:     @echo ""
00197:     @echo ":-) Completed!"
00198:     @echo ""
```

图 32.8

其中的更改点有：

（1）第 84 行定义了 MKDIR 变量，用于记录创建目录所使用的命令，在这里是“mkdir”。

（2）第 83 行定义的 MKDIRFLAGS 变量被用于存放目录创建时所使用的命令参数。

（3）第 84 行将$(DIR_PROFILE)目录放入 RMS 变量中，以便在“make clean”时删除。DIR_PROFILE 变量是在第 88 行定义的，它的值为“profile”。

（4）第 90 行增加了 profile 假目标。

（5）第 185～188 行用于检查单元测试可执行文件是否已构建出来了，这是运行“make

profile”的先决条件。

（6）第 190～191 行是调用 valgrind 中的 callgrind 工具进行性能分析，分析后的结果将以文件的形式存放在 build/unitest 目录中。

（7）第 194～198 行定义了构建 profile 目标的规则。profile 目标依赖于$(UTS)，这将导致第 190～191 行定义的规则被用于对每一个单元测试可执行程序进行性能分析。分析完成后，第 195 行在 build 目录下创建 profile 目录，并将位于 build/ unitest 目录下所生成的性能分析报告拷贝到其中。

32.2.2　检查整合效果

图 32.9 显示了“make profile”运行结束后 build 目录和 profile 子目录下的内容。

```
                        /embedded/build  make unitest
                        /embedded/build  make profile
                        /embedded/build  ls
c.rule  c++.rule  Makefile  profile/  unitest/
                        /embedded/build  ls profile
unitest_bitmap.prof     unitest_clib.prof        unitest_dllht.prof
unitest_dll.prof        unitest_task.prof
```

图 32.9

使用 kcachegrind 打开 unitest_dll.prof 文件，将获得图 32.10 所示的界面。

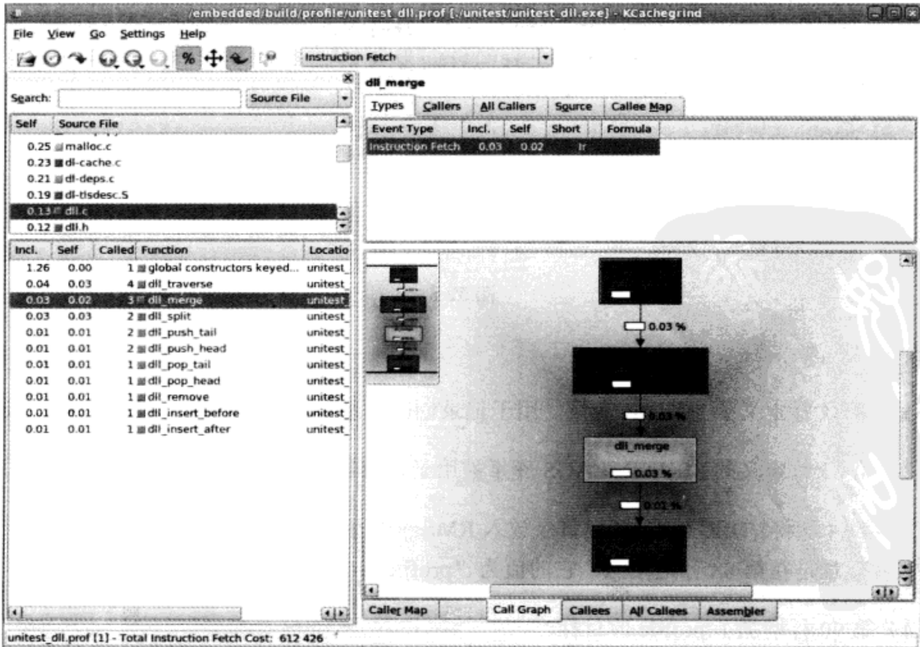


图 32.10

一个性能分析报告中将包含可执行文件中所有函数的分析数据。对于 unitest_dll.prof，我们应当只关注被分析文件 dll.c 中各函数的性能数据。通过在 kcachegrind 中选择 dll.c 文件，可以获得该文件内各函数的处理器时间开销。

32.3　小结

程序性能的优化不应一味地依赖感觉和经验，通过统计数据找到性能瓶颈将显得更加的科学和专业，而性能分析的目的也在于此。

valgrind 除了提供动态分析的功能外，还为我们带来了性能分析的功能。通过将 valgrind 的性能分析功能无缝地整合到开发环境中，使得我们又多了一个便利的开发利器。



第 33 章 qBench，一个开发高质软件的工作台

作者希望 embedded 项目所使用到的、有助于保证软件质量的方法能被读者运用到工作中，也希望本书所构建的 embedded 项目的开发环境能成为读者的工作台——一个打造高质量软件的工作台。为此，为整个开发环境取了“qBench”一名，它是“Quality Bench”的缩写。下面我们通过回顾的形式，看一看 qBench 中包含什么内容，并以本章结束质量保证篇。

设计是软件产品质量之本。在 embedded 项目中，我们运用了统一的错误码定义方法（第 15 章）来管理错误；也通过引入分层和分级的概念，使用统一的模块管理方法实现了模块的有序初始化和终止化（第 14 章）；当然，embedded 项目中模块的设计也运用了不少设计原则（第 13 章）以使软件更加的专业、简单和优雅；等等。尽管设计比较抽象，但作者还是倾向于将 embedded 项目中所有与设计相关的、有助于提高软件质量的内容纳入到 qBench 的框架之下。

像项目目录结构规划（第 16 章）这样的“小事”对于软件质量也很重要。好的目录结构能反映功能性、折射层次感和体现模块化。embedded 项目使用了分层和分模块的组织方式管理所有的源文件，软件开发无小事，目录结构的规划同样应包含在 qBench 之中。

再好的软件设计也需要通过代码去表达，软件的功能也是通过程序去实现的。因此，保证代码质量是软件质量保证活动中很重要的课题。质量保证离不开系统性的方法论，方法论体现在软件开发的各个阶段部署恰当、够用的流程和工具。就代码质量的保证来说，流程与工具需要与开发环境做到无缝整合，只有这样工程师才乐于使用而不会只将它们当做摆设。在 embedded 项目中，通过以单元测试为中心，将单元测试、代码覆盖、静态分析、动态分析和性能分析无缝地整合到了开发环境中，使得工程师可以通过运行简单的 make 命令将这些方法运用于保证代码质量。毫无疑问，这些被无缝整合的方法也应当归到 qBench 的旗下。

尽管这里宣称 qBench 是一个高质量软件的开发工作台，但我们还是不能忘记软件质量保证是一个很复杂的系统工程，也不能忘记软件行业“没有银弹”。每一个项目团队都应当根据自己的实际情况，不断地根据项目开发活动中的薄弱环节修正自己的开发流程。流程的运用离不开工具的部署，即使部署工具的初衷是好的，但也千万不要将重点从软件开发活动本身转移

到工具上。一味地追求某一工具被高效地使用，很有可能是走入了“救命稻草”这一死胡同。

高质量软件的获得虽然需要恰当的方法论，但合适的人永远是关键中的关键。项目团队只有拥有整合流程与工具能力的人，才有可能构建与自身匹配的方法论；也只有项目团队拥有了掌握设计方法和思想的人，才有可能设计出出色的软件架构；高质量代码的获得还需要工程师具备良好的工作习惯，这同样与人紧密相关；等等。

身处软件行业的我们，对于“唯一不会变化的就是变化本身”这句话应当有非常深刻的认识。我们只有不断地通过学习和实践去提高自己的技能，并通过更科学的方法保证自己的工作质量，才能更从容地面对工作中的变化，以及找到工作与生活的平衡点。



参考资料

[1] **Brennan's Guide to Inline Assembly**

Brennan "Bas" Underwood

[2] **Choosing the best real-time scheduling method for your embedded device**

Kerry Johnson, Jason Clarke, Paul Leroux, and Robert Craig, QNX Software Systems

http://www.eetimes.com/design/embedded/4006631/Choosing-the-best-real-time-scheduling-method-for-your-embedded-device?Ecosystem=embedded

[3] **Deadline Scheduling**

Peter Dibble

http://www.eetimes.com/design/embedded/4023324/Deadline-Scheduling

[4] **Debugging with gdb (The gnu Source-Level Debugger)**

Richard Stallman, Roland Pesch, Stan Shebs, et al.

[5] **GNU Make (A Program for Directing Recompilation)**

Richard M. Stallman, Roland McGrath, Paul D. Smith

[6] **Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 1 Basic Architecture**

http://www.intel.com/products/processor/manuals/

[7] **Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2A Instruction Set Reference（A-M）**

http://www.intel.com/products/processor/manuals/

[8] **Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2B Instruction Set Reference（N-Z）**

http://www.intel.com/products/processor/manuals/

[9] **Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 3A System Programming Guide**

http://www.intel.com/products/processor/manuals/

[10] **Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 3B System Programming Guide**

http://www.intel.com/products/processor/manuals/

[11] **Intel x86 Function-call Conventions - Assembly View**

http://www.unixwiz.net/techtips/win32-callconv-asm.html

[12] **Introduction to Rate Monotonic Scheduling**

David Stewart, Michael Barr

http://www.eetimes.com/discussion/beginner-s-corner/4023927/Introduction-to-Rate-Monotonic-Scheduling

[13] **Keeping your priorities straight: Part 1 - context switching**

William E. Lamie

http://www.eetimes.com/design/embedded/4008216/Keeping-your-priorities-straight-Part-1--context-switching/

[14] **Multitasking alternatives and the perils of preemption**

Michael Barr

http://www.eetimes.com/design/embedded/4006706/Multitasking-alternatives-and-the-perils-of-preemption

[15] **The GNU Linker (ld)**

Steve Chamberlain, Ian Lance Taylor

[16] **Using as (The GNU Assembler)**

Dean Elsner, Jay Fenlason & friends

[17] **Using the GNU Compiler Collection**

Richard M. Stallman and the GCC Developer Community



[18] **What Every Programmer Should Know About Memory**

Ulrich Drepper

[19] **What Is Software Design?**

Jack W. Reeves

http://www.developerdotstar.com/mag/articles/reeves_design.html

[20] **人月神话**

Frederick P. Brooks

清华大学出版社

[21] **C 语言参考手册**

Samuel P. Harbison III, Guy L. Steele Jr.

机械工业出版社

[22] **C 陷阱与缺陷**

Andrew Koening

人民邮电出版社

[23] **C 专家编程**

Peter Van Der Linden

人民邮电出版社

[24] **现代操作系统**

Andrew S. Tanenbaum

机械工业出版社

[25] **嵌入式实时操作系统 uC/OS-II**

Jean J. Labrosse

北京航空航天大学出版社

[26] **eCOS 实时操作系统源代码**

http://ecos.sourceware.org/

[27] **GLIBC 源代码**

http://www.gnu.org/s/libc/

[28] **RTEMS 实时操作系统源代码**

http://www.rtems.com/



［Ｇｅｎｅｒａｌ Ｉｎｆｏｒｍａｔｉｏｎ］
书名＝专业嵌入式软件开发　全面走向高质高效编程
作者＝李云著
页数＝６２０
出版社＝北京市：电子工业出版社
出版日期＝２０１１．１０
ＳＳ号＝１２８６５９５５
ＤＸ号＝０００００８１９８７３３
ＵＲＬ＝ｈｔｔｐ：／／ｂｏｏｋ．ｓｚｄｎｅｔ．ｏｒｇ．ｃｎ／ｂｏｏｋＤｅｔａｉｌ．ｊｓｐ？ｄｘＮｕｍｂｅｒ＝０００００８１９８７３３＆ｄ＝８４Ｃ６３６９５ＢＦ８Ｂ８６５８７Ｂ００ＢＢ１４ＢＥＤ５４Ｃ９７